DISK ENCLOSED

ON DISK: Keep track of your windows

AC's TECH AMIGA

For The Commodore

Volume 2 Number 4
US $14.95 Canada $19.95

In Search of... the Lost Window

- Putting the Input Device to Work
- Getconfirm[ ]
- Programming the Amiga in Assembly Language, Part 5
- True BASIC Extensions
- Tape Drives
- Fast Plots
- Entropy in Coding Theory
- The Joy of Sets
- Advanced Scripting

PLUS!

- Quarterback 5.0— *a review*

04
0 74470 78640 5

Bring your Amiga to *life!*

AMOS — The Creator is like nothing you've ever seen before on the Amiga. If you want to harness the hidden power of your Amiga, then AMOS is for you!
AMOS Basic is a sophisticated development language with more than 500 different commands to produce the results you want with the minimum of effort. This special version of AMOS has been created to perfectly meet the needs of American Amiga owners. It includes clearer and brighter graphics than ever before, and a specially adapted screen size (NTSC).

"Whether you are a budding Amiga programmer who wants to create fancy graphics without weeks of typing, or a seasoned veteran who wants to build a graphic user interface with the minimum of fuss and link with C routines, AMOS is ideal for you."
Amazing Computing, June 1992

HERE ARE JUST SOME OF THE THINGS YOU CAN DO

- Define and animate hardware and software sprites (bobs) with lightning speed
- Display up to eight screens on your TV at once – each with its own color palette and resolution (including HAM, interlace, half-brite and dual playfield modes)
- Scroll a screen with ease. Create multi-level parallax scrolling by overlapping different screens – perfect for scrolling shoot-em-ups
- Use the unique AMOS Animation Language to create complex animation sequences for sprites, bobs or screens which work on interrupt
- Play Soundtracker, Sonix or GMC (Games Music Creator) tunes or IFF samples on interrupt to bring your programs vividly to life
- Use commands like RAINBOW and COPPER MOVE to create fabulous color bars like the very best demos
- Transfer STOS programs to your Amiga and quickly get them working like the original
- Use AMOS on any Amiga from an A500 with a single drive to the very latest model with hard disc

WHAT YOU GET AMOS Basic, sprite editor, Magic Forest and Amosteroids arcade games, Castle Amos graphical adventure, Number Leap educational game, 400-page manual with more than 80 example programs on disc, sample tunes, sprite files and registration card.

If you've got an Amiga you need AMOS!

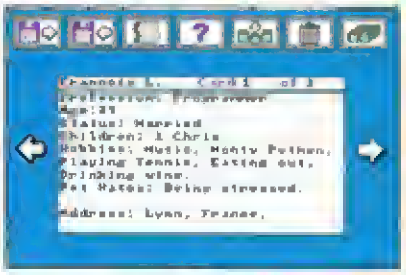
For help you can phone the special US SUPPORT LINE on 219 874 6380
Alternatively you can access the special BBS line fo ON-SCREEN HELP on 219 874 0367

EUROPRESS SOFTWARE

AMOS written by François Lionet.

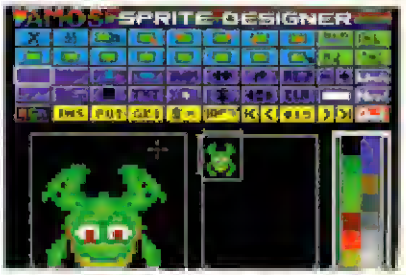
Country of origin: UK

Use the sophisticated editor to design your creations


Create serious software like Dataflex

Produce educational programs with ease


Play Magic Forest and see just what AMOS can do!



Design sprites using the powerful Sprite Editor


Create breathtaking graphical effects as never before

Circle 103 on Reader Service card.

Contents

Volume 2, Number 4 AC's TECH/AMIGA

Departments

```
printf("Hello");

print "Hello"

JSR printMsg

say "Hello"

writeln("Hello")
```

Whatever language you speak, AC's TECH provides a platform for both gaining insight and sharing information on its most innovative implementation for the Amiga. Why not see if your latest programming endeavor can help a fellow Amiga user expand upon his or her vocabulary? To be considered for publication in AC's TECH, submit your technically oriented article (both hard copy & disk) to:

AC's TECH Submissions
PiM Publications, Inc.
One Currant Place
Fall River, MA 02722

AC's TECH AMIGA

ADMINISTRATION

Publisher:	Joyce Hicks
Assistant Publisher:	Robert J. Hicks
Administrative Asst.:	Donna Viveiros
Circulation Manager:	Doris Gamble
Asst. Circulation:	Traci Desmarais
Traffic Manager:	Robert Gamble
Marketing Manager:	Ernest P. Viveiros Sr.

EDITORIAL

Managing Editor:	Don Hicks
Editor:	Jeffrey Gamble.
Hardware Editor:	Ernest P. Viveiros Sr
Senior Copy Editor:	Paul Larrivee
Copy Editor:	Timothy Duarte
Video Consultant:	Frank McMahon
Art Director:	Richard Hess
Illustrator:	Brian Fox
Editorial Assistant:	Torrey Adams

ADVERTISING SALES

Advertising Manager:	Wayne Amuda
Advertising Associate:	Edward McKenney

1-508-678-4200
1-800-345-3360
FAX 1-508-675-6002

AC's TECH For The Commodore Amiga™ (ISSN 1053-7929) is published quarterly by PiM Publications, Inc., One Currant Road, P.O. Box 2140, Fall River, MA 02722-2140.

Subscriptions in the U.S., 4 issues for $44.95, in Canada & Mexico surface, $52.95, foreign surface for $56.95.

Second-Class postage paid at Fall River, MA 02722.

POSTMASTER: Send address changes to PiM Publications Inc., P.O. Box 2140, Fall River, MA 02722-2140. Printed in the U.S.A.

First Class or Air Mail rates available upon request. PiM Publications, Inc. maintains the right to refuse any advertising.

PiM Publications Inc. is not obligated to return unsolicited materials. All requested returns must be received with a Self Addressed Stamped Mailer.

Send article submissions in both manuscript and disk format with your name, address, telephone, and Social Security Number on each to the Editor. Requests for Author's Guides should be directed to the address listed above.

Startup-Sequence

Development

This magazine was originally created for the high end user and developer. So far, it has been successful in reaching its intended audience. I want to improve on that primary goal by setting a new agenda for *AC's TECH*.

It can be frustrating at times to watch the Amiga market. Ami sales could always be better. Product development is sometimes slow, and the computer community's attitude toward the Amiga is often negative. The Amiga itself is a good platform. And like any other computer, with the right combination of hardware, software, and ingenuity, it can become the perfect machine for whatever task it is asked to perform.

Further research and development is needed to advance the Amiga and make it a leading platform in the 1990's. Hardware and software development need to complement each other. The Amiga has a lot of potential. The boom in video development has proven just that. There are, however, a great deal of Amiga users who do not use their Amigas for video production. The development community must address the game players, desktop publishers, and home and business productivity people as well as the video users.

I don't want this editorial to come across as one of those "feel-good-hail-Amiga" pieces. The readers of this magazine already have reasons to like or dislike their Amigas and don't need me to play salesperson. Instead, my aim is to motivate the Amiga programmer, developer, and software publisher into creating higher quality software for this machine. We cannot wait for Microsoft or Aldus to port their products over to the Amiga. Therefore, we have no option but to develop equal or better products. This is not to say that the software available at this time is below standard—it's not. But there is always room for improvement.

AC's TECH should be used as a venue for new development ideas.

I am going to ask that you send us any suggestions for articles which you would like to see. Also, if you have an article you were thinking of writing for us, please go ahead and do so. I want you to sit down at your Amiga and write code that will blow away anything created before it. Think of a problem with your favorite piece of Amiga software and find a solution to it. If the solution is to generate a new piece of software, then by all means, create it. The more development, the more products. The more products, the higher the competition. The higher the competition, the higher the quality. High quality software and hardware is what we all want and need.

Take the initiative and start those creative juices flowing.

Submissions

We receive hundreds of submissions for *AC's TECH* throughout the year. They are all well written and informative, but unfortunately we cannot publish all of them. Therefore, we look for the strongest articles—the ones that will bring the most benefit to our readers. If you choose to send an article to us, make sure it will stand out from the rest. Present a well written, informative, interesting, and beneficial article. Give the reader a reason to purchase the magazine and read your article. We also need small utilities to include on the *AC's TECH* disk. If you have something, send your program and all necessary files along with 1-2 pages of documentation to our submissions department.

Part of the new agenda for *AC's TECH* will be to spark an interest in creating software and hardware for the Amiga. The magazine has been directed at the high-end user. It will continue to fulfill the needs of the high-end user, programmer, and developer. We will also start to nourish the creativity of those of us who do not have a great deal of development experience. High-level users must grow from beginners. I want *AC's TECH* to be used as a tool. A tool for learning. A tool for development. A tool for creativity.

Together we can improve the quality of computing life for all Amiga users.

Jeff Gamble
Editor

Putting the Input Device to Work

by David A. Blackwell

Introduction

Almost every program requires some form of input and output. Programs that are totally self-contained, lacking any form of interaction, are exceptionally boring and prone to quick obsolescence. In fact, I am unable to think of any programs off hand that do not have some form of input or output, regardless of the source of input or the destination of the output.

On the Amiga, your input and output options are very flexible. You can choose from more than a dozen ways to perform I/O using the Amiga operating system and hardware. The type of program you design determines the I/O methods best suited for your purposes. If you are writing a program that you want to be compatible with future releases of the Amiga operating system and hardware, you will select the system acceptable I/O methods. If you are not concerned with future compatibility with system software or hardware, you can select I/O methods that perhaps even include direct access to various hardware registers. Our focus in this article will be basic user input and output using the lowest level of system acceptable I/O — The input device.

Comparison

Before describing how to write your own input device handler, we should compare the three acceptable I/O methods for basic user input and output. You will be better prepared to choose which I/O method, or collection of methods, is best for you after reading this comparison. We will start with the Console Device.

Console Device

During normal system operation, the console device is situated at the top of the input event food chain. Essentially, what this means is that the console device gets all the input scraps. What the other users (handlers) of the input event chain don't consume gets passed to the console device. However, if your window is the active window, and the console device is your only source of input and output, you will receive just about any input event you desire. On the other hand, if you have more sources of input than just the console device, your console device can be completely shut out by the other devices. That could be your design though, and you may be planning to use the console device simply for it output capabilities.

The real strength of the console device lies in its output capabilities. The console device has the most extensive list of text oriented output commands of the three acceptable I/O methods. You will most likely choose the console device if you need total control of the screen text display. The I/O process is somewhat involved, however, and requires a lot of messages be passed between the program and the Console Device. You use messages to request input as well as perform output. On the other hand, the easiest type of input method has to be Intuition using the Intuition Direct Communications Message Port (IDCMP).

IDCMP

Using an IDCMP is the easiest way to connect your program to the input event stream. Intuition does most of the setup work for you. Intuition knows you want it to setup an IDCMP for you when the IDCMPFlags field of the NewWindow structure is not NULL. Intuition will then automatically create a message port for you. You specify which input events you want to receive by selecting the appropriate IDCMP flag bits (List One contains the IDCMP flags).

This is just a brief introduction to the IDCMP. For a complete description of how to get the IDCMP to work for you, read "Programming The Amiga's GUI" in C" by Paul Castonguay in AC's TECH volume 2, number 3. He does a very thorough job and has promised to continue with more insight into the IDCMP.

The main drawback of using an IDCMP is the lack of output. An IDCMP is essentially an input only method of user I/O. You will most likely use the console device, graphics library or a combination of the two when you need to output anything. For detailed text control you use the console device. For basic text functions and drawing images and lines on the window you use the graphics library. Intuition passes any input events that it does not need to the console device; however, it receives its input from the input device.

Input Device

As I stated earlier, the input device is the lowest level of acceptable system input and output. By this I mean that Intuition and the console device receive their input events from the input device. The input device, however, is not the absolutely lowest level of system I/O. The input device receives its input from the keyboard, timer and gameport devices. As programmers though, the lowest level of I/O recommended for us is the input device. So, to use the correct programming parlance, the keyboard, timer and gameport devices are "feeders" of the input device, and Intuition and the console device are "users" or "handlers" of the output of the input device (no pun intended).

Handlers of the input-stream — output of the input device — can be at different levels within the input-stream. These various levels, also called priority positions, range from -128 to +127. The higher priority handlers sit closer to the source of the input. Intuition is at position 50, and the console device is at position 0. That shows why Intuition gets first pick at the input events and the console device gets the leftovers.

As its name implies, the input device is mostly useful for receiving input; however, it can be used to generate output. As a method of output though, it is quite inefficient. You perform output by allocating InputEvent structures for each character you want output. Then you have to allocate an IOStdReq structure. You pass the IOStdReq structure to the input device requesting it to add the InputEvent structures to the input-stream. You can see that this is a clumsy method of output. Like with the IDCMP, you would most likely use the console device or the graphics library to handle your output.

Unlike using an IDCMP or the console device, you are not required to have a window to receive input from the input device. With an IDCMP or the console device, not only must you have a window, but your window must be the currently active window in order to receive input.

This aspect of using the input device is extremely useful if you want your program to be able to manipulate other windows such as perform cut and paste type operations. Your window can go to the back while activating other windows. Then by monitoring the input-stream for specific key strokes, you can perform operations in any window in the system. Another possibility is closing your window completely and allowing your program to sleep in the background. Then you wait for a special combination of keys to be pressed. By closing your display window you free up chip memory for system use, and you reduce the workload on the CPU. When your special key combination is pressed, your program comes back to life, opens its display window and awaits input from the user.

These brief descriptions of the three methods of basic system I/O should help you determine which method is what you need. Now we are going to take a more detailed look at the input device and the advantages to its use.

Let's Get Started

The input device is just like any other device in the Amiga operating system. You must open it before you can use it. You send I/O requests to it to perform various functions. When you are finished using the input device, you must close it.

Two steps are required before calling the OpenDevice() function to open the input device: create a message port that the input device can use as a reply port, create the necessary structures to pass commands and data to the input device. To use the input device effectively you will have to be familiar with several structures the input device uses to do its processing.

The IOStdReq and timerequest structures are used to pass commands and data to the input device. The input device recognizes four of the nine standard Exec commands and eight device specific commands (List Two contains the complete list of input device commands). The IOStdReq structure is used for the majority of the commands while the timerequest structure is used for the IND_SETTHRESH and IND_SETPERIOD commands. I include a short description of each command in the InputDevice.h header file.

You use the CreateExtIO() function to allocate the structures mentioned above. The arguments of this function include the message port address of the reply port and the size of the structure to be allocated. The following two example lines of code will allocate an IOStdReq structure and a timerequest structure.

```
CreateExtIO(ReplyPort, sizeof(struct IOStdReq));
CreateExtIO(ReplyPort, sizeof(struct timerequest));
```

Of course you will want to check to make sure the CreateExtIO() function does not return an error. Once this structure is allocated you can open the input device. The following function call will accomplish this.

```
OpenDevice("input.device", NULL, IORequest, NULL);
```

The first NULL represents a device number. This will almost always be NULL unless you are using the trackdisk or timer device. The IORequest is your IOStdReq or timerequest structure that you previously allocated. The last NULL represents flag bits. The flag bits indicate options for some devices. The input device does not accept options in this manner so this is always set to NULL. A zero return from the OpenDevice() function indicates success. The last structure you need to be aware of is the InputEvent structure.

The main structure that the input device uses to pass information to input-stream handlers is the InputEvent structure. The input device uses the InputEvent structure to pass information on keypresses, mouse movements, gadget events, etc. down the input-stream.

When you are done using the input device you close it. To do this you call the Exec CloseDevice() function passing a pointer to a IOStdReq or timerequest structure as an argument. This will be the same structure that you used to open the device.

I just lightly touched on using the input device in this manner. Our main emphasis in this article is writing our own input device handler.

Input Handler

At first, what may appear to be a daunting task turns out to be surprisingly simple; as long as you know assembly language. Because every clock cycle your Input Handler uses reduces the remaining time before the next event happens, I highly recommend that you write your handler in assembly language.

The AC Tech articles written by William P. Nee covering assembly language are worthwhile to read if you are unfamiliar with assembly language programming. I also recommend Motorola's book on the 68000 series of microprocessors (published by Prentice Hall). This book contains the execution times of the 68000 instruction set to help you write the quickest code. Although this book contains a complete listing of the instruction sets of various 68000 series microprocessors, it does not teach assembly language programming. When writing your handler code, you will need to know what you will be receiving as input.

When your handler is called, address register A0 holds a pointer to a linked list of input events, and address register A1 holds a pointer to your handler's data area. The CPU has eight address registers (A0-A7), six of which are available for your program to use (A0-A5). Register A6 holds the base address of the shared library you are currently using, and register A7 holds the stack pointer. The list of input events is a single-linked list. This means you can only go through it in one direction, and when the pointer to the next event is NULL, you have reached the end.

Your handler's data area is whatever you want it to be. In the example program, I use this data area to pass information to my handler that it will need to operate. I also use this area to communicate with my handler from my C program. Before the input device can tell your handler where this data area is, you must first tell the input device where it is.

You will pass the address of your data area in an Interrupt structure. This structure will also hold the entry address to your handler code. I will go into this in greater detail later in the article. For now we will get back to writing our handler.

Since address registers A0 and A1 are scratch registers, and their values can be changed by any system routine you might call, it is a good idea to find a safe place to put the values passed in them. In my code, I chose the next two address registers, A2 and A3, for this task. Since these registers are not scratch registers, I had to save their values on the stack so I could restore them when my handler was done. I use the following code to handle do this.

```
movem.l A2-A3, (sp) ;saves non-scratch registers
movea.l A0,A2 ;put event list pointer in A2
movea.l A1,A3 ;put handler data pointer in A3
```

Now A2 will hold the pointer to the current inputevent structure I am working on and A3 holds the pointer to my data area throughout the execution of my handler.

Since these are pointer values, it is easy to access the particular fields I need by using the indexed indirect addressing mode. This is similar to the pointer addressing used by C. An example will help illustrate this point. In C, I would use the statement

```
myData->hd_msgsize = 20;
```

to assign the value 20 to the hd_msgsize field in the myData structure. In assembly language

```
move.l #20,hd_msgsize(A3)
```

would do the same thing. The C pointer variable myData and the address register A3 hold the starting address to the block of memory that represents the structure that holds the data area for my handler. The hd_msgsize offset value is added to the pointer address to make it point to the proper location in memory to store the given value. The C structure member hd_msgsize and the assembler constant hd_msgsize are not the same label. Although they look exactly alike and actually represent the same offset value, they are given their respective values at two different locations in the program. The C preprocessor assigns the offset value to the hd_msgsize structure member when it encounters the structure definition,

```
Struct myHandlerData {
struct MsgPort *hd_Port;
ULONG hd_msgsize;
ULONG hd_memreqs;
ULONG hd_passevents;
};
```

in the InputDevice.h header file. The assembly language constant receives its value when the assembler encounters the assembler directive,

```
hd_msgsize equ $0
```

in the InputDevice.i header file. If you look in the InputDevice.i header file, you will notice other assembly language constants that look exactly like their C language counterparts. I named them that way for consistency. In your own program you can use any names you wish. Besides the recommendation to write your handler in assembly language for speed, there are other rules that you need to be aware of while writing your handler code.

Handler Rules

Your handler may allocate memory to use to communicate with another program through a message port or to use to add new events to the linked list it passes to lower priority handlers. It can actually allocate memory for any purpose it wants but these are the most common uses.

Allocating memory for messages and copying information from the inputevent structures in the linked list is preferred to having your handler do the processing of the input events you are looking for. Your handler should request a block of memory the size you need by using the Exec AllocMem() function. This can be done with the following code:

```
move.l hd_msgsize(A3),D0 ;size of message structure
move.l hd_memreqs(A3),D1 ;request to clear memory
jsr -198(A6) ;AllocMem(size, requirements)
movea.l D0,A1 ;put address of allocated memory in
A1
tst.l D0 ;see if I actually received some memory
```

In the first two statements I am setting up to make the call to the AllocMem() function. I determined which registers where used by this function by looking in the Exec pragma file that came in the include files with my C compiler. Then I looked at the function prototype to see what value each register represented. The third statement calls the AllocMem() function.

When your handler code is called, the A6 register holds the base address of the Exec shared library. You can access any Exec function by providing the correct negative offset from this base address. Again I looked in the Exec pragma file to obtain this offset. It is given as a hexadecimal number right before the function name. In this case the value was given as 0xC6. I took this value and converted it to a decimal value. Then I put a negative sign in front of the decimal

number and thus I had the proper negative offset for the AllocMem() function.

The very next statement after my call to allocate some memory puts the returned memory address in the A1 address register. At this point you may be wondering how I knew the address of the allocated memory was in the data register D0. This is very simple. In the Amiga operating system, any function that returns a value, returns that value in the D0 data register. Again, the CPU has eight data registers (D0-D7) and all eight are available for your use. However, only data registers D0 and D1 are considered scratch registers. If you plan to use any of the other data registers in your code, you should save their values on the stack at the beginning of your routine so you can restore their values when your routine exits.

As with any other routine, the AllocMem() function may not be able to accomplish your request successfully. That is why the fourth statement in the example above tests the contents of the D0 register with the tst.l instruction. This is an easy way to test for a NULL value which in this case would signify an error. This instruction is normally followed by a branching instruction to handle an error condition.

After I am sure I have my message memory, I copy the information I need from the inputevent structure into my message structure, and I send the message to my awaiting C program. I used the Exec PutMsg() function to send the message. Again I used the function pragma and prototype to determine register usage and proper negative offset from the Exec base address. In my C program, I process the message and free the message memory when I am done with it. This is the method you should also use. This processing load would slow the input handler down to an unacceptable speed and would slow the entire system down to a crawl. That is generally not a good idea.

After some additional processing, I check to see if there is another inputevent structure to process. If so, I go back to allocate some more memory and send another message. If not, I restore the values of any non-scratch address and data registers I may have used and then exit. My code that does this is as follows:

```
move.l io_NextEvent(A2),D0 ;get next event
movea.l D0,A2 ;put event structure pointer in A2
bne.s more ;go back if there is another event
movem.l (sp)+,A2-A3 ;restore non-scratch registers
move.l (sp)+,D0 ;put event list pointer in D0
rts
```

The first three instructions determine if another inputevent structure is there to process. The bne.s instruction goes back if there are more, otherwise, the handler exits. The next instruction after the bne.s instruction restores the values of the non-scratch registers. The next instruction's purpose may not be so clear. The very first thing I did when my handler was called was to save the address of the first inputevent structure on the stack. The output of the handler is supposed to be a new linked list in the D0 data register. Since I never physically remove any events from the list, I just pull the first address off the stack directly into the D0 register and exit my handler routine.

As previously stated, your handler is free to add events to the linked list of events also. To do this you must also allocate memory for the inputevent structure. You then initialize the structure with the values you desire and add it to the list. The next time your handler is called you are free to return the memory your inputevent structure used to the free memory pool or reuse it for another inputevent. The important thing to remember is that your handler is responsible for keeping track of any memory allocated for this purpose and eventually returning it to the free memory pool.

Another rule concerning input handlers is that they are free to unlink any input event in the linked list they receive. In fact, you are encouraged to unlink events because that means that you will pass a shorter list to the lower priority handlers. This speeds up the overall process of servicing the input event stream. In fact, if you remove every event structure and return a NULL value in D0, the input device will not even call the lower priority handlers. I chose a slightly different method of event removal though. Instead of physically removing the event, I converted it to a NULL class of event. This action produces the same effect and reduces the complexity of my handler code.

That describes the rules your handler code should follow and provides you with example handler code to study. I would like to caution you on one point. Programming bugs in your handler routine are very difficult to debug. Make every effort to insure that your handler code is bug free before you submit it to the input device. There is another issue you will have to take care of first; making your handler code entry point available to your C program.

Because your handler code is written in assembly language, your C program does not really have any knowledge of its existence. This creates a problem when you have to supply the handler entry address to the input device. There are two statements you must include to make your C program aware of your handler's existence. In your C code include the

```
extern void myEventHandler(void);
```

prototype, of course change the handler name to whatever you call your routine. Next, in your assembly language section of your program include the

```
xdef _myEventHandler
```

assembler directive. Then make sure the _myEventhandler: label is at the beginning of your handler routine. The reason for the initial underscore character is that C appends an underscore to the beginning of all its variables and function names. If you didn't do it, then the names would not match when C assigned the underscore to the external myEventhandler() function you declared in your prototype. The linker would then be unable to fill in a value whenever you referenced the function in your C code and would report an error at link time. Being aware of these practices of the C compiler saves a great deal of headaches.

Once you have written your handler code using assembly language to make it as quick as possible, and you made its entry point available to your C code, you are ready to add it to the list of input device handlers. Now we are back working in C so this process is fairly simple.

Adding Your Handler

The first thing you are going to do is allocate the memory for the structures you are going to need. Then you open the input device exactly as described above. In addition to the IOStdReq structure you will need an Interrupt structure.

You use the Exec AllocMem() function to get the memory you will use for the Interrupt structure. In the Interrupt structure you are going to set up your priority, the name of your interrupt routine, the pointer to the handler data area, and the pointer to the entry point of your handler routine. These values are placed in the is_Node.ln_Pri, is_Node.ln_Name, is_Data and is_Code fields of the Interrupt structure respectively. You setup your IOStdReq structure after you finish with the Interrupt structure.

In the IOStdReq structure you set up the command to pass to the input device — in this instance it will be the IND_ADDHANDLER command — and the pointer to your Interrupt structure. These values are placed in the io_Command and the io_Data fields of the IOStdReq structure respectively. Your set up is now all done and you are ready to add your input handler to the list of handlers.

Your next action is to pass your IOStdReq structure to the input device. There are two functions available to accomplish this task. They are the DoIO() and SendIO() functions. The only difference is that one operates synchronously while the other operates asynchronously. I

chose the synchronous function, DoIO(), because I did not want my program to continue until I was sure my I/O request had been completed.

Once your I/O request has been successfully executed by the input device, your handler is hard at work processing input events. This is also where you will most likely find out if there are any bugs in your handler code. Expect a quick GURU visit if there are problems with your handler code. Hopefully you won't encounter any problems and your handler will execute perfectly.

Closing Down

The only point remaining after you get your handler working properly is shutting it down. It is important to make sure that you return all the memory you allocated to the free memory pool.

Your first order of business is to shut down the input handler. Since we used global variables for the Interrupt and IOStdReq structure we used to add the input handler, they are still available for use. All that we need do now is change the command in the IOStdReq structure from IND_ADDHANDLER TO IND_REMHANDLER. Once this is done, send the IOStdReq structure to the input device. Once the input handler is shut done you can close the input device.

Now you are free to release the memory used for the Interrupt and IOStdReq structures. You use the FreeMem() function to release the Interrupt structure memory and the DeleteExtIO() function to release the IOStdReq structure memory.

As you receive and process any last messages your handler sent before being shut down you will free up any memory used by them. You can wait until you exit the program to close you message port if you want to.

That pretty much outlines what you need to know to put an input event handler to work for you. By going over the example program closely, you will get better insight into how each of the three I/O methods described work together. You will especially see how the input handler works and how to get it to communicate with your C program.

Example Program

I wrote the example program to provide instruction on writing your own input device handler. I did not really intend to teach assembly language programming. I touched the topic a little in the description of the handler code, but I do expect some familiarity with assembly language programming. It is essential when you begin to write your own handler. I believe that it is a good idea to have a foundation in assembly language as well as an intermediate or high level language. The example program, besides showing you how to write, install, handle data from and close down your own input device handler, shows how an input device handler can be used in conjunction with the other I/O options available to you in the Amiga operating system.

With a little modification, you can begin to see how the different I/O systems work together. In the install_input_handler() function you can change the priority of the handler. This changes your handler's relative position within the inputstream. This can provide some insight in how your handler can insert input events in the linked list at the proper level. This can be done to ensure that the input event is processed by the handler you have it in mind for. There are various ways you can experiment with this example program code to explore different I/O setups.

Program Operation

The example program opens a screen and a backdrop window for its display. In the top portion of the window are three boxes that display the input events received by the input device handler, the IDCMP and the console device. Below the display boxes are four gadgets. Three gadgets are labeled as Input Device Gadget, IDCMP Gadget and Key Type Gadget. The last gadget is the QUIT gadget.

The first two gadgets mentioned control the flow of keyboard events to the lower priority handlers. The input device handler defaults to passing the events without modification. When you click on the gadget it changes. Then the input device will consume all keyboard events. When you tell the input device handler to consume keyboard events it changes the event class to a NULL class. This makes the lower priority handlers pass over these events.

The IDCMP gadget operates in the same manner as the Input Device Gadget except that it defaults to consuming the keyboard events. The steps taken to consume or pass the events are different, but the outcome is the same. The difference between the way the input device handler passes input events and the way the IDCMP passes input events is that the input device handler still receives the input event that it later passes while the IDCMP never receives the event. You will notice this in the three display boxes as you run the example program.

The QUIT gadget is self-explanatory I believe. Use this gadget only when you have grown tired of seeing how the three I/O methods work together.

The last item on the screen is the message area at the very bottom. Hopefully, you will not get to see this function in action. This area is only used to report problems encountered while the program is running.

Conclusion

The example program contains much more useful information within its code than I was able to put into writing in this article. I recommend that you look over the code to see how its works. As always, I hope that this program helps you gain experience with an aspect of the Amiga's operating system that you were unfamiliar with. I enjoy receiving your comments, recommendations and suggestions. For those readers who subscribe to GEnie, you can send me your comments using the e-mail address, D.Blackwell1.

All the source code and executable files for this article may be found on the AC's TECH disk.

Please write to:
David Blackwell
c/o Amazing Computing
P.O. Box 2140
Fall River, MA 02722

In Search of... the Lost Window

by Phil Burk

The windows on the computer screen were designed to simulate the papers on our desk. This metaphor works very well, but is sometimes too close to reality. Documents, letters, notes, and program listings accumulate until I feel like an archaeologist excavating the layers of an ancient civilization whenever I try to find something on my desk. I can date a document by how many inches it is buried below the top surface.

My Amiga's Workbench can also become cluttered when I have several programs running, each with multiple windows. I became tired of sifting through these windows with the push-back gadgets so I wrote a simple program called FindWindow. With this program, I can now get to any window on the Workbench with one simple action. I thought it would be marginally useful, so I gave it to a few friends. Much to my surprise, they use it constantly, and it has earned a permanent spot on their Workbench screens. FindWindow opens a very small window that usually sits in the Workbench title bar. When you click on the window with the *left* mouse button, a popup menu appears with a list of all open Workbench windows. You can select a window which will pop to the front and be activated. Selecting from a popup menu using the *left* mousebutton may seem strange at first. I chose it because using the *right* button would require two mouse actions—one with the left button to activate the small window and a second action with the right button to use the menu.

The program was written using *JForth Professional Version 3* available from Delta Research. If you are programming in C or any other fine language, I trust the principles in this program can be applied to your own work.

ONE-MINUTE FORTH LESSON

Here is a quick Forth lesson to help you read and understand the program listing. Forth has a large dictionary of commands called "words." Forth words typically operate on a stack of numbers. For example, the plus operator '+' removes the top two numbers from the stack, adds them together, and places the sum on the stack. The print operator '.', pronounced "dot," removes the top number from the stack and prints it. Comments are in parentheses. For example:

```
200 123 + .   ( would print 323 )
```

This system is called Reverse Polish Notation, or RPN. It is the style used by HP calculators. It may seem a bit odd at first, but is quite easy to learn because everything happens in a simple left to right order.

To define a new word and add it to the dictionary, Forth uses a colon ':', followed by the name of the new word. A "stack diagram" usually follows that describes what is expected on the stack and what is left after execution. A stack diagram that uses curly brackets '{' defines local variables similar to C. The word definition is ended with a semicolon. For example, here is a word called SUMSQ that uses local variables and calculates the sum of the squares of two input values.

```
: SUMSQ  { a b - a2+b2 , calc sum of squares
   a a *  ( calculate square of A )
   b b *  ( square B, both squares are on stack )
   +  ( add together top two numbers on stack )
  This could be tested by entering:

  4 SUMSQ .       which would print "25".
```

PROGRAM OVERVIEW

Now let's dig into the FindWindow program and see how it works. Here is a pseudo-code description of the program:

> Read x,y and color parameters from command line.
> Open small window and draw buttons.

```
Wait for mouse events.
If mouse button pressed in "Windows" box then
        Scan Window List for Window titles,
        Open Popup Menu with titles,
        Wait for Mouse up to select window,
        Bring selectd window to front and activate,
        Close Popup Menu.
If mouse button pressed in close box then
        Close window and exit.
```

The small window contains three buttons, a close box, an about box, and a button for the popup menu. I did not use Intuition Gadgets for these because I could tell which button was hit simply by looking at the X-position of the mouse. Gadgets are worth using for more complex applications, but this technique often provides a simpler solution. The code that analyzes the X-position is in the word FW.PROCESS in Find_Window.f.

Most of the program is fairly straightforward and can be deciphered from the comments. There are two areas, however, which need some explanation: scanning the windows for their titles, and the popup menu.

GETTING THE WINDOW TITLES

The most important trick in FindWindow is getting the titles of all the open windows. On the Amiga, the first place to look for information is usually in the Library Base structures. When you open a library you get an address to a Library Base structure. Below this address in memory are the jumps to the various functions in the library. Above this address is a structure which contains lots of information used by the library.

In the Intuition Library, there is a pointer to the Active Screen. Since we will be using the Workbench, that will be the Active Screen. Every screen has a pointer to its first window. Each window has a pointer to the next window in the screen. Here is a summary of what points to what:

```
IntuitionBase -> ActiveScreen -> FirstWindow -> NextWindow
```

Let's use JForth to interactively explore these structures. This is often the way a JForth programmer will work. First figure out how the Amiga works by interactively experimenting. Once the technique is understood, then it is time to write the code. If you have JForth, fire it up. If not, follow along and we'll explain things as we go.

First we should compile some tools to help us. DUMP_STRUCT will print the names and contents of the members of a structure. VALUE is a handy type of variable. Enter in JForth:

```
include ju:dump_struct include ju:value
```

After 2 or 3 seconds, JForth will print how many bytes were added to the dictionary. Now let's attach the precompiled include files that contain the Amiga structure definitions. Enter:

```
getmodule intuition
```

To open the Intuition library in JForth, we can use INTUITION?. This will place the address of the IntuitionBase in a variable called INTUITION_LIB. We can fetch the address using @ then declare a value called IBASE that we can use to refer to the structure by name. Enter in JForth:

```
intuition?      intuition_lib @ >rel value IBASE
```

IBASE contains the address of the IntuitionBase structure. To dump out the contents of the IntuitionBase, enter:

```
ibase dst IntuitionBase
```

Here is a partial listing of the result:

```
652,088    S4   IB_ACTIVEWINDOW
582,320    S4   IB_ACTIVESCREEN <  ::!
582,320    S4   IB_FIRSTSCREEN
 73,'40    U4   IB_FLAGS
    329    S2   IB_MOUSEY
```

Notice that the Mouse X,Y-position is stored in this structure. Just for fun, try moving the mouse and then reprinting the structure to see the numbers change. We could write to this structure using the S! operator. For more fun than most people can stand, push the mouse over to the left side of the screen and enter this:

```
250 ibase s! ib_MouseX
```

Then touch the mouse and watch the cursor jump! Please note that the IntuitionBase is considered PRIVATE and READONLY by Commodore. Do not write to this structure in any of the programs that you develop.
The IntuitionBase structure contains a Library structure which in turn contains a Node. To see these structures, enter:

```
ibase dst library
ibase dst node
```

Now let's get back to what we're supposed to be doing. The IntuitionBase also contains the address of the currently Active Screen. We can fetch from this structure member using the S@ operator.

```
ibase s@ ib_ActiveScreen  value ASCR    ascr dst screen
( dump screen structure )
```

From the screen, we can get the address of the first window. We can then fetch the title of the window and print it. The word 0COUNT, pronounced "zero count," will take a C-style NUL terminated string and return the length of the string and the address of the first character which are needed by TYPE. Enter:

```
ascr s@ sc_FirstWindow value MYWIND     mywind      sw
wd_title 0count type
```

On my system, the first window title printed was:

```
Amiga_DOS_Shell
```

Each window contains a pointer to the next window. The last window contains a zero. Thus, we can scan this list in a loop until we get a zero window pointer. Let's get the next window. Enter:

```
mywind @ wd_NextWindow  > mywind        mywind
.     Is it zero?
mywind @ wd_title
  0count type
```

Since the Amiga is a multi-tasking system, it is possible for windows to be opening and closing while we scan the list of windows. When scanning any system lists you should first call Forbid() and Permit(). We did not do this in our interactive example because it is impolite to Forbid() other tasks for too long a time.

To see how FindWindow uses the above technique to get the window names, see the following words in the listings:

```
GET.ACTIVE.SCREEN in Popup_Menus.f
FW.DRAW.NAMES in Find_Window.f
```

FindWindow filters out windows with blank titles, BACKDROP windows, and its own windows.

POPUP MENUS

I wanted the user to be able to place the program's main window anywhere on the screen and have the menu pop up right over that window. I also wanted to use the *left* mouse button. Thus, I could not use Intuition Menus. I decided to implement a popup menu toolbox as a stand-alone toolbox so that it could be used with other programs as well.

To make a menu pop up, you would call POPUP.OPEN with the x,y-position, width, number of items, and an optional title. This just opens a BORDERLESS window. If the window extends beyond the edges of the screen, it will be moved to fit. This is determined by looking at the screen width and height in (POPUP.OPEN).

You can then draw the text for the menu items by calling POPUP.DRAW.TEXT with a text string and an item. It is assumed at this point that the mouse button is still down, because a mouse down triggered the menu to pop up. We now call POPUP.SCAN which tracks the mouse, highlighting items as needed until the mouse button is released. POPUP.SCAN returns the item number chosen. To remove the popup menu, call POPUP.CLOSE.

COMPILING FindWindow

You can type in the source code for FindWindow or download it from GENIE in the Amiga section. It is item number 14135.

JForth has the ability to create a small stand-alone application from a compiled application. To compile FindWindow and to create a stand-alone application, enter in the CLI:


```
RUN C:JForth INCLUDE make_fw
```

To test the program, enter in JForth or the CLI:

```
FindWindow -X 260 -F 2
```

You can modify and test the program directly from JForth without cloning it. To make this a permanent part of your Workbench, copy FindWindow to your C: directory; then RUN it from your startup sequence.

I hope this program will be useful to you. If you have comments or questions, or would like to get the FindWindow program and source on a floppy disk, please feel free to contact me.

All source code and executable files for In Search of the Lost Window can be found on the AC's TECH Disk.

Please write to:
Phil Burke
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

AC Disks

Source code and executable programs included for all articles printed in *Amazing Computing.*

11

AC V5.8, V5.9 and AC V5.10
Fully Utilizing the 68881 Math Coprocessor Part III:
Fittings and Turbo_Pixel Function. Author: Read Predmore.

C Notes From the C Group 5.8 & 5.10: Functions supporting doubly linked lists, and a program that will examine an archive file and remove any files that have been extracted. Author: Stephen Kemp.

Time Out!: Accessing the Amiga's system timer device via Modula-2. Author: Mark Cashman.

Stock/Portfolio: A program to organize and track investments, music libraries, mailing lists, etc. in AmigaBASIC. Author: G. L. Penrose.

CygCC: An ARexx programming tutorial. Author: Duncan Thomson.

Programming in C on a Floppy System: Begin to develop programs in C with just one megabyte of RAM. Author: Paul Miller.

Koch Flakes: Using the preprocessor to organize your programming. Author: Paul Castonguay.

AudioIllusion: Experience an amazing audio illusion generated on the Amiga in Benchmark Modula-2. Author: Craig Zupke.

Pictures: IFF pictures from past Amazing Computing issues.

12

AC V5.11, V5.12 & V6.1
Keyboard Input In Assembly: Fourth in a series of Assembly 68000 programming tutorials. Author: Jeff Glatt.

A Shared Library for Matrix Manipulations: Creating a shared library can be easy. Author: Randy Finch.

C Notes From The C Group: A discussion on cryptography. Author: Stephen Kemp.

ZoomBox: Attaches a zoom box to an Intuition window and allows the user to toggle the window's size and its position. Author: John Leonard.

13

AC V6.2 & V6.3
C Notes 6.2: A reminder program to display messages. Author: Stephen Kemp.

More Ports For Your Amiga: Files to accompany article. Author: Jeff Lavin.

Ultra Sonic Ranging System: BASIC Sonar Ranging program. Author: John Iovine.

Writing Faster Assembly: Continuing the discussion of speeding up programs. Author: Martin F. Combs.

C Notes 6.3: Working with functions. Author: Stephen Kemp.

14

AC V6.4 and V6.5
Blitz Basic: Here are some examples created with M.A.S.T.'s integrated BASIC environment. Author: Paul Castonguay.

Creative And Time-Saving Techniques: Enchancing and fine-tuning images through definition. Part of the Fractal series. Author: Paul Castonguay.

Fractal Modula-2 Buffered Disk I/O: Buffer file input and output to improve disk accessing speed. Author: Michal Todorovic.

15

AC V6.6, V6.7, V6.8, & V6.9
Practicalities: Practical uses of Finch's previously documented Matrix.library. Author: Randy Finch.

Selecting and Setting Gadgets In C: The third and final installment in the "Crunchy Frog" approach to programming. Author: Jim Frost.

C Notes6.6: A new skeletal program to "jump start" utility programs. Author: Stephen Kemp.

Fancy Numbers: This helps you save overhead by skipping the translator library. Author: Lynwood Cowan.

C Notes6.7: Adding functions to handle file pattern processing. Author: Stephen Kemp.

Message Logger: A time log that keeps track of when programs are run. Author: Brian Zupke.

Power Basic: Use a pre-processor to achieve definition replacement. Author: Jonathan Horne.

16

AC 6.10, 6.11, 6.12, 7.1, 7.2, 7.3
Puzzled Over ARexx Parts 1&2: ARexx have you running around in circles? Learn the usage of basic commands through this entry-level ARexx program. Author: Merrill Callaway.

Simplified File Decompression Using ARexx: Compress and decompress files with this simple ARexx program. Author: Randy Finch.

Jump Tables in Modula-2: Learn the intricate details of Modula-2 programming. Author: Michal Todorovic.

DePuzzle: With this neat little program you can solve some age-old probability questions. Author: Scott Palmer.

ZipTerm: An explanation of console device and serial device on the Amiga. Author: Doug Thain.

ARexx Translator: The premier of AC's ARexx column. Using ARexx to translate number bases and character codes. Author: Merrill Callaway.

FC Calc: Create these MaxiPlan templates to help organize interest and finance charges on your credit cards. Author: Rick Manasa.

Recursive Function Calls in ARexx: How to create and use recursive function calls in ARexx. Author: Merrill Callaway.

Getconfirm()

A Dynamic Requester Function

by John Baez

Usually, in the course of executing some command selected by the application user, we encounter conditions in which it becomes necessary to prompt the user for a response. I'm referring to requesters such as those which prompt for things like: "Exit without saving. Are you sure?" or maybe "Bad response entered. Retry." You may also want to present a more extensive textual description such as a help display. It is somewhat cumbersome and unproductive to build custom requesters for these situations. Intuition does provide the AutoRequest() function. Sometimes, however, you don't need two responses and sometimes you may need more. Another drawback of AutoRequest() is that you cannot select your response from the keyboard. GetConfirm() is a handy little function which adjusts itself dynamically to the situation at hand. It provides a very simple way to get a simple response from the user or to just make him aware of, and confirm, an event (hence, the name GetConfirm).

Arguments of GetConfirm()

Table I lists and explains the arguments of GetConfirm(). Listing 1 is a sample program which shows some variations of GetConfirm(). As you can see by running the sample program, GetConfirm() returns a 1, 2, or 3 which represents the gadget which was selected. If the arguments are bad, a 0 is returned.

You should note that the prompt text is a string terminated with two nulls. GetConfirm() uses two successive nulls to indicate that the prompt string is complete. That is why it is important to insert a '\0' as the last character of the prompt string. You may also embed a single '\0' value anywhere within the text where you want to start a new line, as seen in line 23 of Listing 1.

The user can select an option by clicking on a gadget with the mouse or by pressing the '1', '2', '3', or RETURN keys which correspond to each possible option. RETURN corresponds to option '1' which is considered the default. To get a feel for user interaction with GetConfirm(), compile the program as follows, using SAS/Lattice C):

```
lc1 -r0 -ccekmsw -idf0: -odf0: df0:testconfirm
lc2 -v -s -odf0: df0:testconfirm
blink testconfirm from LIB:c.o+df0:testconfirm.o to
        df0:testconfirm lib LIB:lc.lib+LIB:amiga.lib
SD ND SC
```

Listing 2 shows an implementation of the editfunctions used in my prior article, "Keyboard I/O from Amiga Windows," which uses GetConfirm(). Use this code as a substitute for the editfunctions.c code in that article to see a more practical implementation of GetConfirm().

Listing 3 illustrates the implementation of GetConfirm(). An IntuiText structure is defined for each possible text string (lines 40-44), and three gadget structures are defined, one for each of the possible gadgets (lines 49-55).

The gadgets are first disconnected by insuring that the NextGadget field of the Gadget structure—see Amiga headers or Intuition reference manual for the Gadget structure definition—is null (line 59). We then proceed to see how many of the gadget text strings point to a string (lines 66 to 84).

For each gadget string provided we increment num_gadgets and initialize the appropriate IntuiText structure with the text pointer. As we proceed through each gadget, we check for the largest text string. The length of the largest text string determines the width of all the gadgets (line 92).

The gadget's width is used in the next step to dynamically size the gadget's border (lines 96-99). Finally, the location of the gadgets is determined (lines 101-114). This is done by first checking the type flag to see if the gadget placement is across the bottom of the window or down the right edge. The gadgets are then distributed evenly across the width or height of the window.

The window is then opened using the support window() function. This was introduced in a prior article and is reproduced here in Listing 4 for completeness. The prompt string is then displayed (lines 126-132), and the gadgets are drawn into the window (lines 136-137). We then enter the main event loop (lines 142-154) and wait for a GADGETUP event or a VANILLAKEY. In the case of the VANILLAKEY we check the key pressed to insure that it is one of the valid keys ('1', '2', '3', or RETURN). When a valid response is received, we exit the loop and return the response.

Table I. GetConfirm() arguments

ARGUMENT	DESCRIPTION
sc	A pointer to the Screen structure on which the requester will be displayed.
top	The pixel coordinate at which the top of the requester will start.
left	The pixel coordinate at which the left edge of the requester will start.
width	The width, in pixels, of the requester window.
height	The height, in pixels, of the requester window.
type	The type of gadget placement. A TRUE value means across the bottom; a FALSE or NULL value means down the right side.
itext	A pointer to the prompt string.
tk1	A pointer to the text string for the first gadget. This value cannot be null.
tk2	A pointer to the text string for the second gadget. This pointer may be null if only one option gadget is desired.
tk3	A pointer to the text string for the third gadget. This pointer may be null if only two option gadgets are desired.

Enhancing GetConfirm()

There are a few things we can do to GetConfirm() to make it fancier. One that comes to mind is the capability to display an Intuition Image as well as the prompt text. You can add the capability to display any number of gadgets by passing an array of pointers to strings. Of course, this feature would require the caller to perform additional preparation before calling GetConfirm(). Remember that the original goal of GetConfirm() is to provide a simple way to create a requester, so keep in mind any enhancements that you make.

A more challenging feature would be to change the option gadgets to an Image rather than a Border. This would require that you resize the image dynamically based on text length. Doing this isn't as difficult as it may sound and would make the requester look more refined. You can probably dream up a few things you may want to add to the code. But the idea behind a function like GetConfirm() is to let you concentrate on your application and use it as needed to prompt the application user. Placed into a shared library, it becomes particularly attractive to users of simple languages like AmigaBasic. In a future article we'll build a "tools" shared library which will contain GetConfirm() as well as a few other goodies.

Listing One

```
*
* This file contains the source code for the app..ration Operatio
* edit function.  You should always #include it before defining
* the IOField structure, which is referenced in
*
edit_date.w.first.field
struct Window *w;
struct IOField *first, *field;
{
    int m, d,
    char work[8 ,
    static short days[ ] = {31,29,31,30,31,30,31,31,30,31,30,31};

    strcpy(work,field->Data,2);
    work[2]=0;
    m=d_atoi(work,10);

    if(m<1 || m>12)  /* check for a valid month. */
        m=99;
    else
    {   /* check for a valid day. */
        strcpy(work,(char * field->Data+2);
        d=d_atoi(work,10);
        if(d<1 || d>days[m-1])
            m=99;
    }
    if(m==99)
    {   struct IntuiMessage *m;

        GetConfirm(w,NULL,screen,40,-1,-1,0,0,
                   "Date entered is incorrect.  Please re
NULL,"*Retry",1,0);
        /*
         * GetConfirm() causes an old Rawkey A_TYPEWRITER to our main
window.
         * In order to avoid confusion (an endless loop in this case) we
         * clear the message before returning.
         *
         * Please note that doing this is application dependent.  This
message
         * may be required by the main program (to refresh the window,
         * for example - in which case we would not clear it.
         */
        while(m=(struct IntuiMessage * GetMsg(w->UserPort)) ReplyMsg(m);
        return(0);
    }
    return(1);
}
```

```
*****************************************************************
edit_age.w.first.field
struct Window *w;
struct IOField *first, *field;
{
    short age;

    age =(short *)field->Data;

    if(age<1 || age>130)
    {   struct IntuiMessage *m;

        GetConfirm(w,NULL,screen,40,80,400,50,0,
                   "Age entered is incorrect.  It must be from 1
to 130",
                   "Retry",1,0);
        while(m=(struct IntuiMessage * GetMsg(w->UserPort)) ReplyMsg(m);
    }
    return(0);
}
```

Listing Two

```
#include exec/types.h>
/*****************************************************************
 * Copyright (c) by John Baez.                                   *
 *                                                               *
 * This private, non-commercial use only.  May not be used in any *
 * way without prior written consent from the author.            *
 *                                                               *
 *****************************************************************
 *****************************************************************
 * NAME: GetConfirm()   request a confirm option gadget.         *
 * SYNOPSIS: int GetConfirm(screen,top,left,width,height,type,
 *                                  prompt,gtext1,gtext2,gtext3);
 *
 *             Selection is by mouse or keyboard                 *
 *****************************************************************

GetConfirm(scr,top,left,width,height,type,gtext1,gtext2,gtext3)
struct Screen *scr;
short top, left, width, height, type;
char *gtext1,
*gtext2, *gtext3, *prompt;
{   struct Window *w1;
    struct IntuiMessage *m;

    short num_gadgets, i, k;

    /*
     * Border
     */
    static short coords[2*(MAX_NUM_GADGETS+1)] = {0,0, 14,0, 14,14,
                                                  0,14, 0,0};
    static struct Border b = {0,0,3,0,JAM1,13,coords,0};

    /*
     * Gadget text structures - 1 for prompt text.
     */
    static struct IntuiText
            t0={1,0,JAM1,5,5,NULL,0,0},
            t1={1,0,JAM1,5,5,NULL,0,0},
            t2={1,0,JAM1,5,5,NULL,0,0},
            t3={1,0,JAM1,5,5,NULL,0,0};

    /*
     * Gadget definitions.
     */
    static struct Gadget
                    g1={0,0,0,0,GADGHCOMP,RELVERIFY,BOOLGADGET,

    &BTTR(&b,&t1),0,0,0,0,
                    g2={0,0,0,0,0,GADGHCOMP,RELVERIFY,BOOLGADGET

    &BTTR(&b,&t2),0,0,0,0,
                    g3={0,0,0,0,16,GADGHCOMP,RELVERIFY,BOOLGADGET,

    &BTTR(&b,&t3),0,0,0,0};
    /*
     * Make sure the gadgets are not linked into a list.
     */
```

```
g1.NextGadget=g2.NextGadget=g3.NextGadget=0;

num_gadgets=0;
 /*
  * Find out how many gadgets we need and start setting up the
  * appropriate gadget structures
  */
if(t1[x])
   { num_gadgets=1;
     x=strlen(t1);
     t1.IText=t1;
     if(t2)
        { num_gadgets=2;
          if(strlen(t2)>x) /* Is this gadget the biggest so far? */
             x=strlen(t2);  /* use the bigger gadget size       */
          t2.IText=t2;
          g1.NextGadget=&g2;
          if(t3)
             { num_gadgets=3;
               if(strlen(t3)>x) /* Is this gadget the biggest... */
                  [illegible]
                  x=strlen(t3);   /* Use the biggest gadget size. */
                  [illegible]
               t3.IText=t3;
               g2.NextGadget=&g3;
             }
        }
   }
else /* No gadgets means bad args... so we exit */
   return [illegible];

 /*
  * Calculate our gadget width based on the largest text string
  * we received.
  */
   x=[illegible]x8+10;
 /*
  * Set the gadget borders to the right size.
  */
coords[0]=coords[2]=coords[8]=x;
coords[10]=x-1;
coords[20]=coords[22]=x+2;
g1.Width=g2.Width=g3.Width=x;

if(type) /* Are gadgets to be setup in a row across the window? */
{  k=(width-(x*num_gadgets))/(num_gadgets+1);
   g1.LeftEdge=k;
   g2.LeftEdge=g1.LeftEdge+coords[2]+k;
   g3.LeftEdge=g2.LeftEdge+coords[2]+k;
   g1.TopEdge=g2.TopEdge=g3.TopEdge=height-25;
}
else       /* or in a column down the window? */
{  k=(height-((16*num_gadgets)))/(num_gadgets+1);
   g1.TopEdge=k;
   g2.TopEdge=g1.TopEdge+16+k;
   g3.TopEdge=g2.TopEdge+16+k;
   g1.LeftEdge=g2.LeftEdge=g3.LeftEdge=width-x-16;
}

 /*
  * Open the window.
  */
wd=(struct Window *)
        window(sc,0,top,left,width,height,0,1,
                         SMART_REFRESH|NOCAREREFRESH|ACTIVATE);
 /*
  * Display our prompt string. The prompt string is broken into lines
  * by '\n'. The end of the string is determined when '\0' is found
  */
k=10;
while(*text)
{  t1.IText=text;
   PrintIText(wd->RPort,&t1,10,k);
   text+=strlen(text)+1;
   k+=10;
}
 /*
  * Display gadgets.
  */
AddGList(wd,&g1,0,num_gadgets,0);
RefreshGadgets(&g1,wd,0);

 /*
  * Wait for a gadget to be clicked or equivalent keyboard input
  */
while(1)
{  m=(struct IntuiMessage *)input(wd,GADGETUP|VANILLAKEY);
   if(m->Class==VANILLAKEY)
   {  if(m->Code>'0' && m->Code<=('0'+num_gadgets) || m->Code==13)
      {  k=(m->Code==13)?1:(m->Code-'0');
         break;
      }
   }
   else
   {  k=((struct Gadget *)m->IAddress)->GadgetID;
      break;
   }
}
CloseWindow(wd);

 /*
  * Return the gadget [illegible]
  */
return (int)k;
```

Listing Three

```
#include <intuition/intuition.h>
/*************************************************************************
 *
 *
 * These are general support routines which can be [illegible]
 * programs to create a window and perform [illegible]
 *
 *
 *************************************************************************/

struct Window
*window(screen,title,left,top,width,height,pena,penb,flags)
struct Screen *screen;
UBYTE *title;
SHORT left,top,width,height;
UBYTE pena,penb;
ULONG flags;
{  static struct NewWindow new_window={[illegible]
   0,NULL,NULL,NULL,NULL,
              "Window",NULL,NULL,[illegible]};

   new_window.LeftEdge=left;
   new_window.TopEdge=top;
   new_window.Width=width;
   new_window.Height=height;
   new_window.DetailPen=pena;
   new_window.BlockPen=penb;
   new_window.Flags=flags;
   new_window.Title=title;
   new_window.Screen=screen;
   if(screen!=NULL)
        if(screen->Flags&[illegible])
           new_window.Type=[illegible];
   return((struct Window *)OpenWindow(&new_window));
}
/*************************************************************************/
struct IntuiMessage *input(w,flags)
struct Window *w;
long flags;
{  static struct IntuiMessage m;
   struct IntuiMessage *message;
   if(flags==0)
      return(NULL);
   if(w->IDCMPFlags!=flags)
      ModifyIDCMP(w,flags);
   FOREVER
   {  if(!(message=(struct IntuiMessage *)GetMsg(w->UserPort)))
      {  Wait(1<<w->UserPort->mp_SigBit);
            message=(struct IntuiMessage *)GetMsg(w->UserPort);
      }
      if(message!=NULL)
      m=*message;
      ReplyMsg(message);
      break;
   }
   return(&m);
}
```

✓

All source code and executable files for Getconfirm can be found on the AC's TECH Disk.

Please write to:
John Baez
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

True BASIC

DEVELOPING PROFESSIONAL LANGUAGE EXTENSIONS FOR TRUE BASIC IN SAS/C

by Paul Castonguay

True BASIC's recent release of "Student Editions" for three platforms, IBM, Mac, and Amiga, at the unbelievably low price of $15, puts the original inventors of the BASIC language, Drs. John Kemeny and Thomas Kurtz, once again at the forefront of computer science by making modern, structured programming available to the largest possible number of people. Combine that with their recent "UNIX Editions," and True BASIC stands a good chance of becoming *the* standard dialect on all computers. In fact, many people are already referring to it as such.

To the intermediate or advanced level programmer on a particular platform, like the Amiga, this presents a new opportunity: to design competitive language extensions or LIBRARIES for a variety of high level tasks, and to publish them as either public domain, shareware, or commercial product. Such utilities can be designed either in True BASIC itself, C, assembly, or in any hybrid combination. If you know enough programming to write reasonable utilities, this may be just the right kind of activity for you. Many of the complex and just plain tedious system requirements of stand-alone applications (like screen and window definitions to name only two) are already solved for you by True BASIC. All your routine may need to do is accept a few arguments, process them in ways that would otherwise be too difficult or time consuming, and then either perform some action or return a final value to True BASIC. By writing such routines you will profit by putting to good use whatever programming skills you have developed so far on the Amiga. It may be just what you need if you have been suffering from a long technical slump. At the same time, by publishing your work you will be helping many others remain excited about the Amiga by making it easier for them to program their machines in BASIC.

This article focuses on the steps required to design a Read_IFF_Image routine, which is written partially in C and partially in True BASIC—a so-called hybrid design. The result is a LIBRARY file that can be used by any programmer, advanced or beginner, to read IFF pictures and brushes from within their own programs, including HAM and Extra-Half_Brite, and at impressive speeds.

BACKGROUND

True BASIC is a well-established, well-supported, ANSI-compliant, platform-independent dialect of BASIC that provides all the high level features around which the language was first developed, features that are sorely lacking in so many others. True BASIC is Kemeny & Kurtz's answer to what BASIC should be like on a microcomputer. It has been so successful that it is presently available for more platforms than

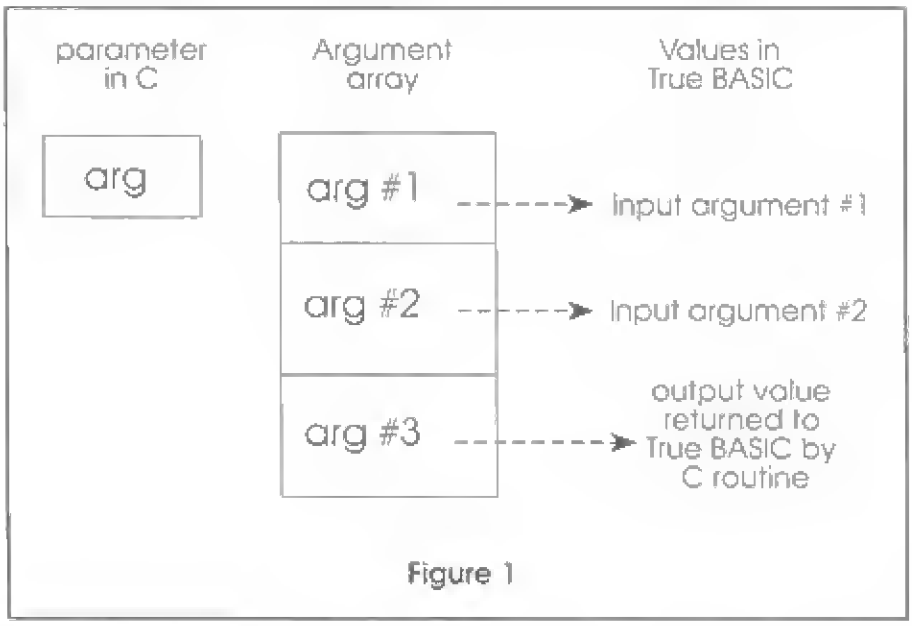


Figure 1

any other single dialect. In fact, it is now being embraced by the UNIX community, as shown by the recent article in the May/June 1992 issue of BASIC-PRO Magazine (available on most newsstands) showing True BASIC being used to design animated CAD applications on the Silicon-Graphics, UNIX workstation. The example code from that article can be run essentially unmodified on any IBM, Mac, or Amiga.

DON'T PUT IT DOWN

Many Amiga users, upon hearing the term platform independence, immediately put down True BASIC, thinking that it will reduce their machine to the level of the most ill-equipped IBM clone. That is simply not so. True BASIC has many powerful and unique features, especially in the area of graphics, that will allow you to get more out of your Amiga. One example is structured drawing, which allows you to easily design complex images in a hierarchical fashion, exactly as recommended by modern programming theory. You will not find that feature in any other dialect of BASIC for the Amiga. Indeed, it is not a part of any other language. In addition, True BASIC provides access to the entire Amiga operating system. It does this in three ways. First, like most programming environments, it allows you to invoke any routine in any of the Amiga's run-time libraries from within your BASIC programs. Second, it provides some high level libraries, like FontLib* in the AmigaTools drawer, which allow you to enjoy certain custom features of the Amiga without having to program it yourself at the level of the ROM Kernel Manual. Finally, and as this article will demonstrate, True BASIC allows you to design your own language extensions or LIBRARIES, thus providing personalized, high level support. The routines within such LIBRARIES can be accessed in two ways, either from the disk files where they are stored, or by merging them into executable memory along with the language itself, a concept called "loading your workspace." Loading LIBRARIES into your workspace makes their routines available to your programs instantly, as if they were a part of the actual language, and without the need to write any LIBRARY instructions within your programs.

You can design a LIBRARY to perform a particular task on one platform, or to meet some more general requirement, in which case your work can be ported essentially unmodified to any other platform supported by True BASIC. But even platform-specific programs, like the one of this article, can be ported with surprising ease. Yes, a very similar LIBRARY can be designed for both the IBM and Mac to read IFF images on them as well. Indeed, it is these kinds of examples that demonstrate, once and for all, that with today's complex operating systems, it is True BASIC that is the most platform-independent language.

HYBRID DESIGNS

Reading IFF pictures requires a reasonable amount of processing power, especially when the graphic data in the BODY chunk of the IFF file is in compressed format, as is the case with pictures and brushes saved by most paint programs. High level languages like BASIC simply cannot perform all the operations required to decode them fast enough. In contrast, C can complete the job in seconds. But C also has some disadvantages. Designing stand-alone applications in C can be very tedious. For example, in my recent series in this magazine called "Programming the Amiga's GUI in C," it took me three articles just to open a graphic screen, something that can be done in True BASIC with a single instruction. Why is C so complicated? Because most implementations are published as professional development packages, where everything that you do requires direct interaction with your computer's operating system, as outlined in the ROM Kernel Manual. Very little, if any, high level support is provided. As a result, you

True BASIC has many powerful and unique features, especially in the area of graphics, that will allow you to get more out of your Amiga.

must design every aspect of your program from the ground up. Programming in such environments may be outside the patience and time constraints of average hobbyists. Indeed, in today's busy world it may be impractical for many professionals as well.

The answer to this performance/complexity dilemma is to design your programs as hybrid packages—partly in C, where program execution is fast, and partly in BASIC, where many standard system operations are fully supported. You won't be able to completely divorce yourself from having to know something about the Amiga's operating system. For example, in order to make the example of this article execute at the fastest possible speed, I had to design it to write graphic data directly to screen memory, something that required an understanding of the Amiga's internal workings. However, by making it a hybrid design, the amount of such direct interaction was greatly reduced. The Read_IFF_Image routine of this article demonstrates the well-balanced use of both low and high level design tools.

I should point out that the new 2.0I operating system has a special library, called iffparse, to help programmers design IFF read and write applications. However, one short peek into the latest ROM Kernel Manual should convince you that, in order to use it, you need to be an advanced programmer. In contrast, the source code of the Read_IFF_Image() routine on this magazine's disk clearly demonstrates that the parsing of an IFF file is much more easily coded in BASIC, without using the iffparse.library. This is especially true if you want to add to your design a few higher level features, like automatic screen resolution switching and error checking.

PRODUCING TRUE BASIC SUBROUTINES IN SAS/C

To produce a True BASIC SUBROUTINE or FUNCTION using SAS/C, you compile your C routine using -b0 for absolute addressing with the first pass compiler program LC1, and -v to suppress stack checking with the second pass program LC2. You also LINK your routine slightly differently. Specifically, you pass the routine's object code through the BLink program, but with the exception that you do not include the usual startup code LIB:c.o. The result is an object file that cannot be executed on its own, but that, with the help of a True BASIC program called FinalTouch, can be converted into a form that is executable from any True BASIC program.

ARGUMENT PASSING

The most important aspect of writing C SUBROUTINES for True BASIC is the communication between the two languages. True BASIC passes values to the C routine in the same way as it does to any SUBROUTINE or FUNCTION, through an argument list. The C routine then accesses them through a parameter called arg, which is a handle (pointer to a pointer) to an array of pointers that represent the arguments that were passed by True BASIC. It is very similar to the way a parameter called argv is used to read command line arguments in AmigaDOS. A second parameter of the C routine, called uv, is a pointer to a structure whose members contain some important system addresses, which can be conveniently borrowed by your C routine. Below is the generic, ANSI prototype definition for a C routine that is intended for use as a SUBROUTINE or FUNCTION from True BASIC.

```
void usr_routine(long **arg, struct TBUserVector *uv);
```

True BASIC provides a header file (in the Assembly drawer of the True BASIC disk) that contains the structure template for the above TBUserVector, as well as structure templates for two types of possible arguments, short integers and strings. Thus, your C code must contain the following inclusion:

```
#include "True BASIC:Assembly/TB.h"
```

You can design your C routine to be either a SUBROUTINE or a FUNCTION. In both cases, values are accessed on the C side

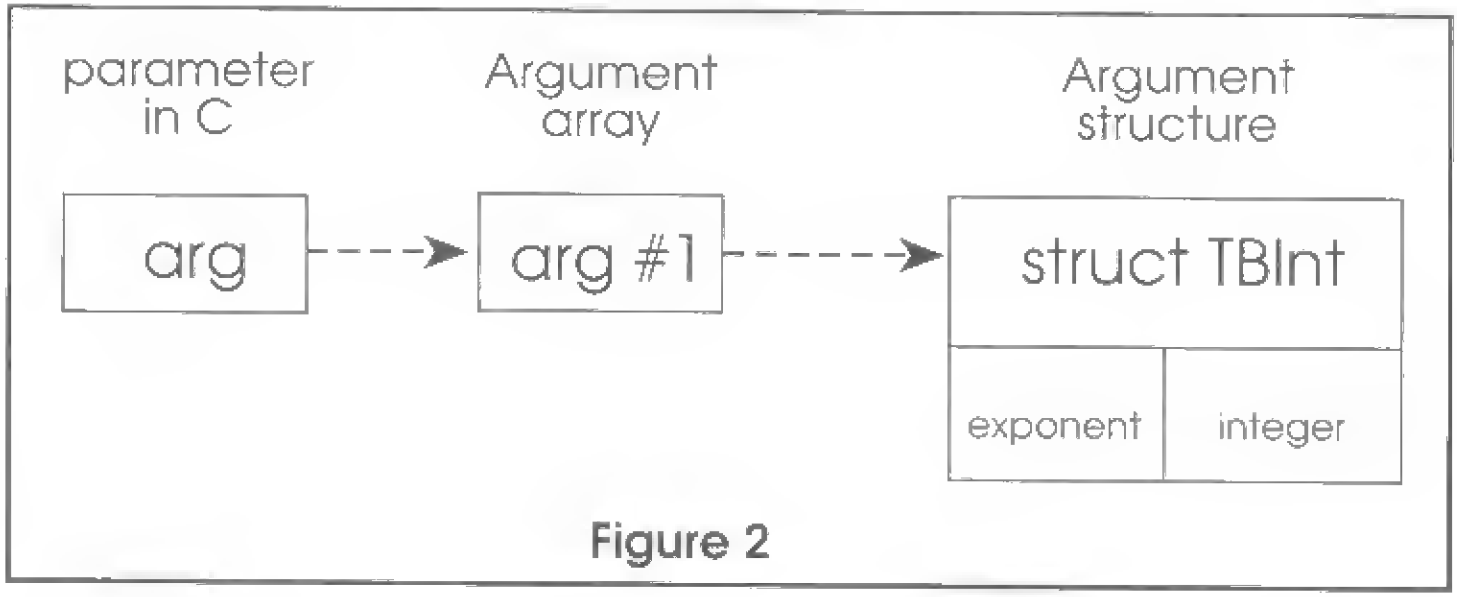


Figure 2

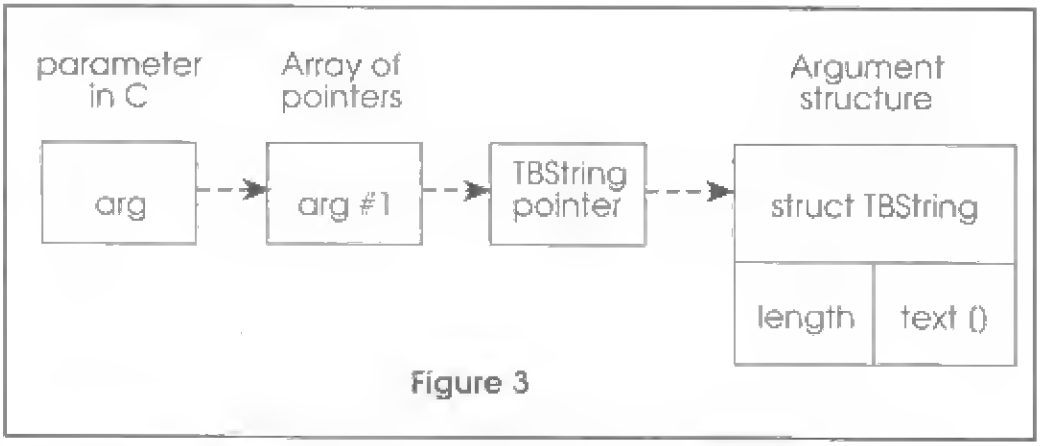


Figure 3

by using addresses passed on the argument array. In the event that the routine is to be a FUNCTION, in which case it will return a value to True BASIC, an extra element is provided at the end of array for that purpose. Figure #1 shows the general concept.

INTEGER ARGUMENTS

Integer arguments are accessed through a special C structure called TBInt, whose template is defined in the header file TB.h. For each element on the argument array that corresponds to a numeric argument, there will exist in memory a TBInt structure that was generated by True BASIC at the time the C routine was invoked. Here is the template definition of that structure as it appears in TB.h.

```
struct TBInt
{
   short exponent;    * flag *
   short integer;     * data *
```

Exponent is a flag that identifies the structure's data as being either type integer or type floating point (-1 means integer). The integer member contains the numeric value itself.

Integers are passed as 16-bit values in 2's complement form, which is equivalent to SAS/C's short integer. Figure #2 shows a SUBROUTINE having one integer argument. Note that the passing of long integers and floating point numbers is done using a slightly different method, which is fully explained in a related doc file on the True BASIC disk.

STRING ARGUMENTS

Strings are accessed through a structure called TBString, whose template is defined in the header file TB.h. For each element on the argument array that corresponds to a string argument, there exists in memory a TBString structure that was generated by True BASIC at the time the routine was invoked. Here is the template definition of that structure as it appears in TB.h.

```
struct TBString
{
   long length;
   char text[1];
}
```

The template shows a character array of only one element. However, the structures that True BASIC generates for your routines will have arrays of whatever lengths are required to pass their arguments, up to a length of one megabyte. Figure #3 shows a SUBROUTINE having a single string argument.

There is a difference here from the previous integer example. The address on the argument array points not directly to a TBString structure, but to a pointer to one. This will affect how the data from a string argument is accessed, as you will soon see.

GENERIC C ROUTINE

With everything that has been said so far, we are led to the following generic form, or model, of a C routine that is intended for use as either a SUBROUTINE or FUNCTION from True BASIC.

```
#include "True BASIC:Assembly/TB.h"

 * proto */
void usr_routine(long **arg, struct TBUserVector
*uv);


void usr_routine(long **arg, struct TBUserVector
*uv)
{
   code to read arguments
      ...

   code of routine
      ...

   code to return a value if a FUNCTION
}
```

ACCESSING THE ARGUMENTS

To access argument values from within the C routine, you must declare pointer variables of the correct types and assign to them the various addresses supplied by True BASIC on the argument array—addresses that represent the argument structures that were created when the routine was invoked. Here is how the addresses of four argument structures, two integers and two strings, can be read and assigned:

```
void usr_routine(long **arg, struct TPUserVector
*uv)

    struct TBInt *arg1, *arg2;

    struct TBString *arg3, *arg4;

    arg1 = (struct TBInt *)*arg ;

    arg2 = (struct TBInt *)*arg ;

    arg3 = (struct TBString *)**arg-;

    arg4 = (struct TBString *)**arg;
```

The above code represents a routine that could be used as a SUBROUTINE from True BASIC in the following way:

```
    CALL usr_routine(X,
Y, A$, B$)
```

The first address on the argument array is read by de-referencing arg and assigning the address obtained to pointer arg1. The post decrement operator then moves arg to the next element position in the argument array. The result is that arg1 points to the TBInt structure corresponding to the first argument passed from True BASIC, which is X. Similarly, arg2 points to the second, Y. Figure #4 shows the result.

For strings I must de-reference arg twice, because their addresses on the argument array represent not the TBString structures themselves, but pointers to them. The above instructions cause arg3 to point to the TBString structure corresponding to the third argument passed from True BASIC, which is A$, and arg4 to point to the last, B$. Figure #5 shows the result.

Pointing to the argument structures is only half the job. To obtain actual values you must read the contents of those structures. Here is how that is done for the above example:

```
int x, y, alen, blen;
char *a, *b;

if(arg1->exponent != -1    arg2 >exponent != -1)
  return;

x   arg1 >integer;

y = arg2->integer;

a    = arg3->text;

alen = arg3->length;

b    = arg4 >text;

blen = arg4->length;
```

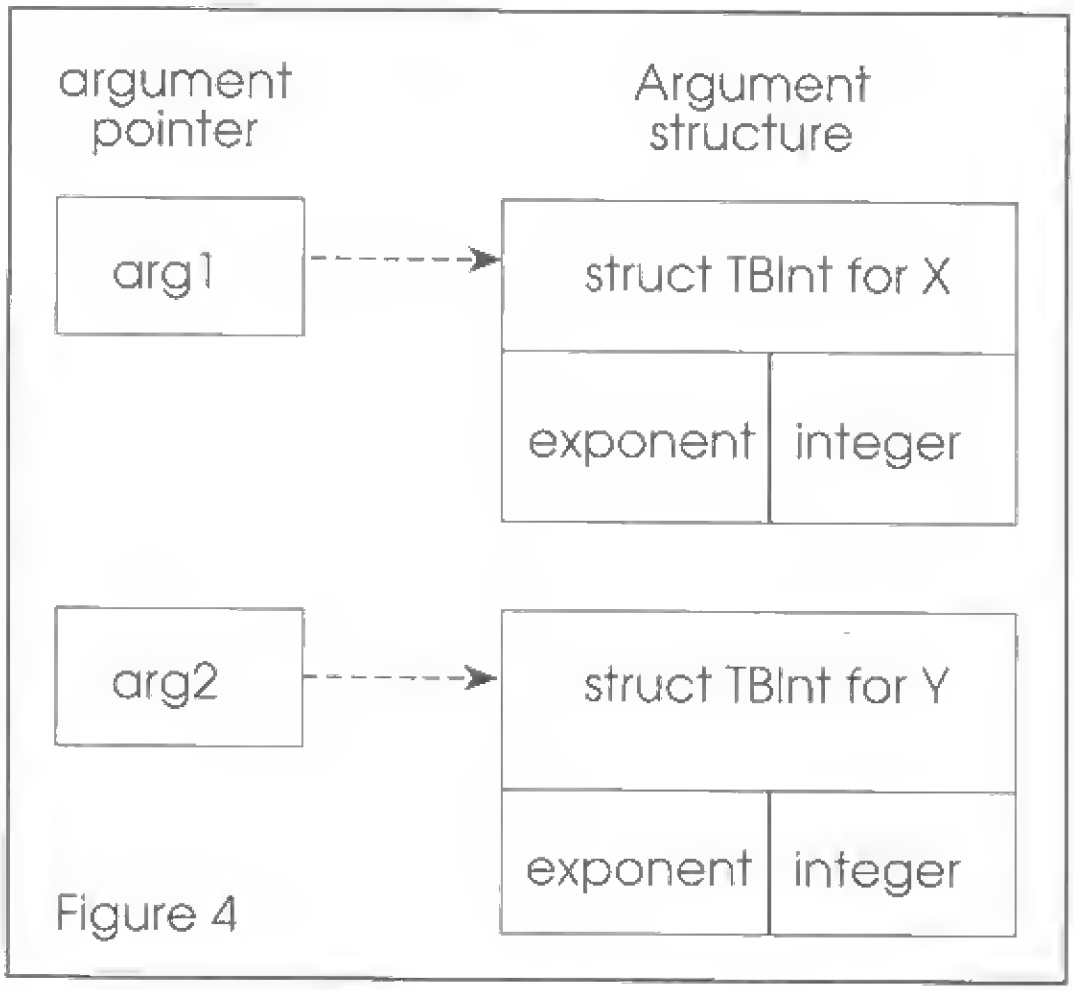


Figure 4

It is prudent to verify that the exponent member of integer arguments is -1. Otherwise, the integer member contains data that is in floating point format, and must be handled differently.

AMIGA SYSTEM POINTERS

To use the Amiga's run-time libraries, a program requires access to their respective system pointers. We could obtain these ourselves by explicitly opening each library, but it's easier to use those already opened by True BASIC. The uv parameter of the C routine is a pointer to a TBUserVector structure, which is defined in the TB.h file, and which contains the addresses of the run-time libraries being used by True BASIC. As you can see below, you also get the address of the window structure that True BASIC is using for your program's output.

```
struct TBUserVector
{
  struct Window *window;  * pointer to our window *
   long DOSBase;          /* DOS library pointer *
  long GfxBase;         /*graphics library pointer *
   long SysBase;          * exec library pointer *
 long IntuitionBase;    /* Intuition library pointer
*
};
```

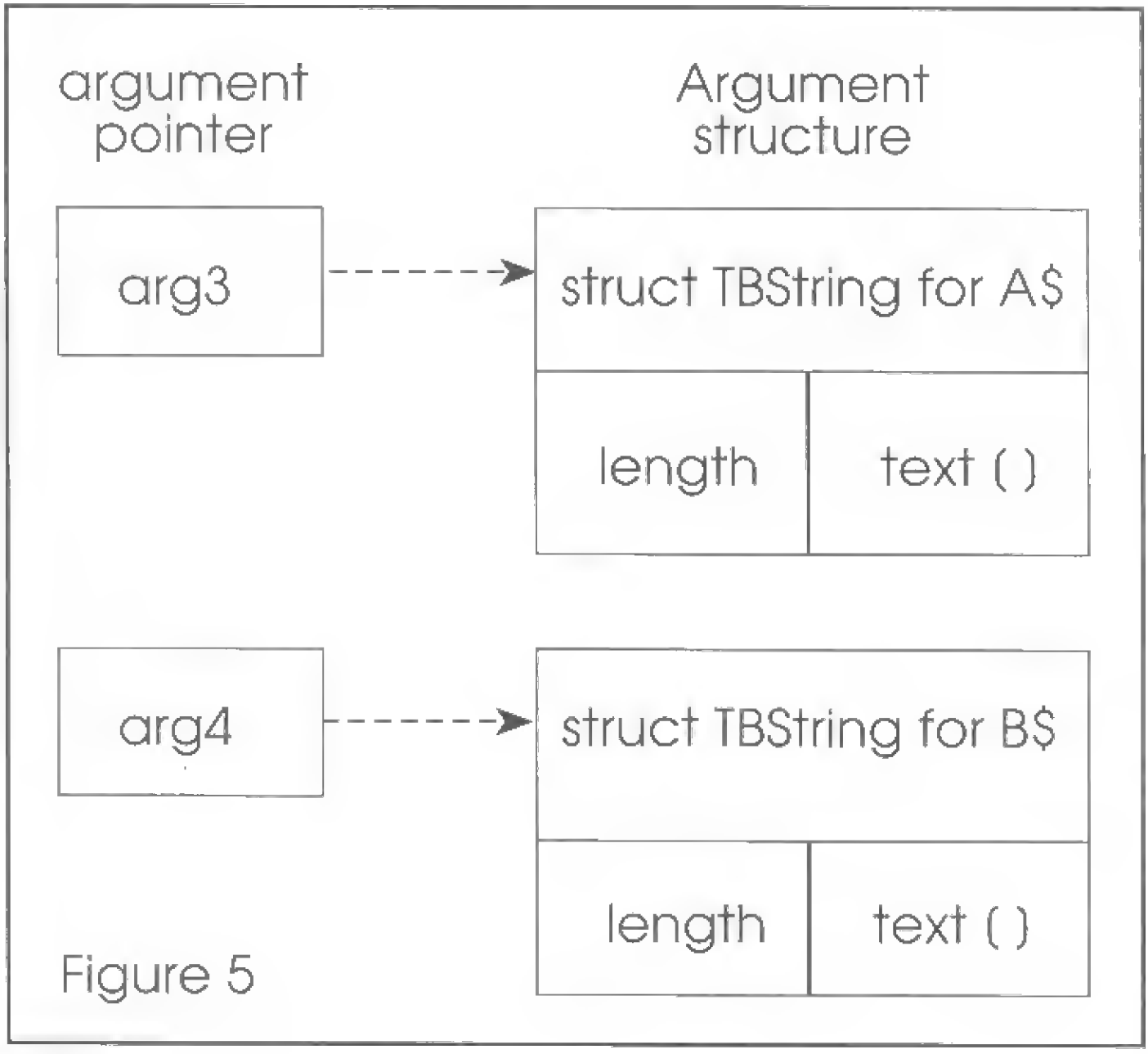


Figure 5

THE DESIGN PLAN FOR Read_IFF_Image

To save development time and allow for easy maintenance, I have designed all sections that do not significantly affect execution time in BASIC, and the sections that require either heavy processing, or that access system structures, in C. The BASIC section, which is invoked by a CALL to Read_IFF_Image, opens a channel to the file, parses it, performs error checking, possibly switches the graphic resolution, and finally invokes the C routine called C___DisplayBody, passing to it a string containing the graphic data read from the BODY chunk of the IFF file, as well as some other related information. Thus, Read_IFF_Image is a driver routine for C___DisplayBody. The triple under-bar in the name is my own way of identifying the nature of this routine and its related files. Of course, you can use whatever notation you like.

THE C PART

The C___DisplayBody.c source code on disk is so well commented that you will probably find it instructional. The first order of business is to declare pointers to the argument structures passed from True BASIC. The first of these is the graphic string. The others represent a variety of related information, like image width, height, depth, position, etc., all of which must be known in order to properly decode and display the image. The program reads each argument structure address, its argument values, and the Amiga system pointers using the techniques I have described above.

To achieve maximum execution speed, I have designed this routine to write the graphic data directly to screen memory. As I mentioned earlier, to do that, I had to understand how the Amiga handled screen memory, and to do that, I had to study the ROM Kernel Manual. My solution defines an array of pointers, BPlane[], each element of which points to an individual bitplane of the current screen. The routine learns about such things as the current screen width, height, and depth from the operating system's Screen and BitMap structures, which are located via the Window address handed over from True BASIC. Following that, my routine enters a data interpretation section where the actual graphic data is decoded and written to screen memory. An interesting feature of my design is that it allows you to position the image on the screen from within True BASIC. Position information is used in the calculations of screen addresses before writing the graphic data to the screen. The routine also clips portions of the image that fall off the right and bottom edges of the screen, preventing it from wrapping around to the left edge, or worse, writing past the limit of screen memory and corrupting the operating system. In addition, the routine discards extra bit planes of the image's graphic data if the current graphic mode is less than that used to create the original image, and if the programmer has suppressed my automatic screen switching feature. As a final protection, the pointer that reads the graphic data, *gdp (graphic data pointer), is constantly checked to make sure that it does not get accidentally incremented beyond the end of the graphic string that was passed from True BASIC. The routine handles both uncompressed data and data that was compressed using the standard ByteRun1 technique described in Commodore's IFF specification.

THE BASIC PART

Although everything performed by Read_IFF_Image in BASIC could theoretically be accomplished faster if it was re-designed in C, the overall speed of the utility would not be improved enough to make the effort of doing so worthwhile. In fact, you might not even notice the difference in overall speed. In addition, and perhaps more importantly, by designing most of the program in BASIC, my work remains open to the widest number of users. In the event that you don't like the way it works, you can easily modify it, even if you are not an advanced programmer. But don't think for one minute that, because the majority of this utility was programmed in BASIC, it is inherently a bad design. On the contrary, the file containing Read_IFF_Image consists of a collection of independent, fundamental program units, like Open_Communication, Read_FORM, Read_BMHD, Read_CMAP, Read_CAMG, etc., all of which get invoked in a hierarchical fashion, exactly as dictated by the most modern, structured programming theory. In addition, all True BASIC routines that are hierarchically subservient to Read_IFF_Image are made PRIVATE. That is, they are made inaccessible from outside the utility. Such structures, called MODULES in True BASIC, prevent certain program units within a utility from being used by you in ways that were not intended. They also protect the routines of your own programs from conflicting with those within the utility if by chance you give one a same name. Lastly, MODULES allow variables to be shared implicitly between a certain number of program units, while at the same time keeping them hidden from the remainder of the project. These are advanced features that most people consider possible only in C or Pascal. They are not only available in True BASIC, they are easier to implement.

Although everything performed by Read_IFF_Image in BASIC could theoretically be accomplished faster if it was re-designed in C, the overall speed of the utility would not be improved enough to make the effort of doing so worthwhile.

When you inspect the True BASIC code, pay attention to the use of exception handling, which produces smart error messages for a wide variety of possible conditions. It is not only a good idea to provide such features in your designs, it is easy to do. I also recommend that the programmer using this utility incorporate exception handling within his/her own programs, as shown in the example code near the end of this

article. This will protect your machine in the event that an unexpected interruption occurs while it is in either HAM or Extra-Half-Brite. You see, although my routine supports these special graphic modes, True BASIC itself officially does not, and it may not recover properly if a program terminates unexpectedly while it is in one of them. It is exactly for situations like these that True BASIC provided its exception handling feature.

Notice how data is read from each chunk of the IFF file using True BASIC's UnPackB() routine, which can read binary data in a bitwise fashion at any location in a string of any length. I take specific advantage of this bitwise feature in routines Ask_If_HAM8 and Ask_If_EHB$. Programmers wanting to do the same thing in C have to design their own routines using bit masking. Lastly, I am particularly proud of the section that automatically switches the graphic resolution, should the programmer want to use that feature. That section would be very difficult and tedious to design in C. In BASIC it was quite easy.

COMPILING THE C CODE

The following LMKFILE, which is used by the SAS/C LMK program to compile your source code, demonstrates exactly how it is done.

```
PROGRAM   DisplayBody
LCFLAGS   -b0 -cfistq -oQUAD:
$(PROGRAM): $(PROGRAM).o
    BLINK FROM $(PROGRAM).o TO $(PROGRAM) LIBRARY
      LIB:lc.lib
    Copy $(PROGRAM) TO $(PROGRAM)*
    COFF $(PROGRAM).info TO $(PROGRAM)*.info

$(PROGRAM).o: $(PROGRAM).c
   LC1 $(LCFLAGS) $(PROGRAM)
   LC2 -v -o$(PROGRAM).o QUAD:$(PROGRAM)
```

Note the options used when passing the C source code through the compiler. The -b0 (32-bit addressing) option must be used with LC1, and -v (no stack checking) with LC2. Without these options your routine will not be accessible from True BASIC. The rest of the stuff is recommended, but not necessary. Now look at how the linker is used. The usual LIB:c.o startup code is not specified. To do so here would produce a self standing, executable program, and we don't want to do that. Instead, we want an object module that will get executed by True BASIC. But first it needs to be converted.

CONVERTING THE C ROUTINE

The program called FinalTouch*, which you will find in the Assembly drawer of your True BASIC disk, is used to convert the object code that was produced by SAS/C using the above LMKFILE into a program unit that can be invoked from any True BASIC program. When you execute the FinalTouch* program it asks you for two entries, the routine's definition, and the name of the file where it is located. To convert this example I entered the following:

```
dec C___displaybody(a$,a,a,a,a,a,a,a)
C___DisplayBody*
```

I designed the C___DisplayBody routine as a FUNCTION to take one string argument, followed by seven integer arguments, and to return an integer that reports success or failure. Thus, to invoke it from True BASIC requires:

```
LET Error = C___DisplayBody(SS,W,H,X,Y,D,M,C)

Where:  SS - Graph...string
        W - Width of IFF image
        H - Height of IFF image
        X - Horizontal position
        Y - Vertical position
        D - Screen depth desired by image
        M - Masking flag
        C - Compression flag
```

But, of course, the people who use this utility within their own BASIC programs don't have to specify all these arguments. That is done for them automatically by the Read_IFF_Image routine.

COMBINING INTO A SINGLE HYBRID DESIGN

At this point the two parts of the utility, BASIC and C, are in separate files. A more convenient approach would be to bind both parts together into a single LIBRARY file. This can be done using True BASIC's MakeLib* utility, which you will find on your True BASIC disk. Figure #6 shows a hierarchical diagram representing the Read_IFF_Image and C___DisplayBody routines. Note, however, that the complete utility encompasses the use of several PRIVATE True BASIC routines, as well as six other C routines, which for brevity's sake I have not mentioned. For that reason a complete hierarchical diagram representing the entire utility would be more complex.

The resulting file, IFF*, can be used as a LIBRARY to effectively add to the True BASIC language the ability to read IFF files, including those in HAM and Extra-Half-Brite. Note that this utility requires the latest version of True BASIC. The reason for this is that I have used some of the language's newer features, like MODULES, which were not a part of the older version.

USING THE UTILITY

To actually read an image and display it on screen for 10 seconds, complete with True BASIC's exception handling feature, you could use the following code:

```
PROGRAM Show_IFF
LIBRARY "AmigaTools:IFF*"
WHEN EXCEPTION IN  ! ------------------------------

    LET Mode$  = "IFF"
    LET Color$ = "IFF"
    LET X = 0
    LET Y = 1
    CALL Read_IFF_Image("Gorilla.IFF", Mode$,
        Color$, X, Y)
```

```
    PAUSE
    CALL Mode_IFF

CASE 2
    CALL Mode_IFF
    EXIT HANDLER
END WHEN
END
```

The Mode$ and Kolor$ arguments are used to control whether you want the routine to automatically switch the current system graphic mode and color definitions to those that were in effect when the original IFF image was created. Making the Mode$ variable equal "IFF" activates automatic mode switching, whereas leaving Mode$ = "" (a NULL string) deactivates it. The Kolor$ argument works the same way. The X and Y variables are used to position the top left corner of the image on the screen. Remember that True BASIC defaults to Normalized Device Coordinates (a 1x1 grid with the reference at the bottom left corner). My above selection of values places the image at the top left corner of the screen. Naturally, if you re-scale the screen these numbers will have to change. An error is produced if you use coordinates that place the top left corner of the image off the screen.

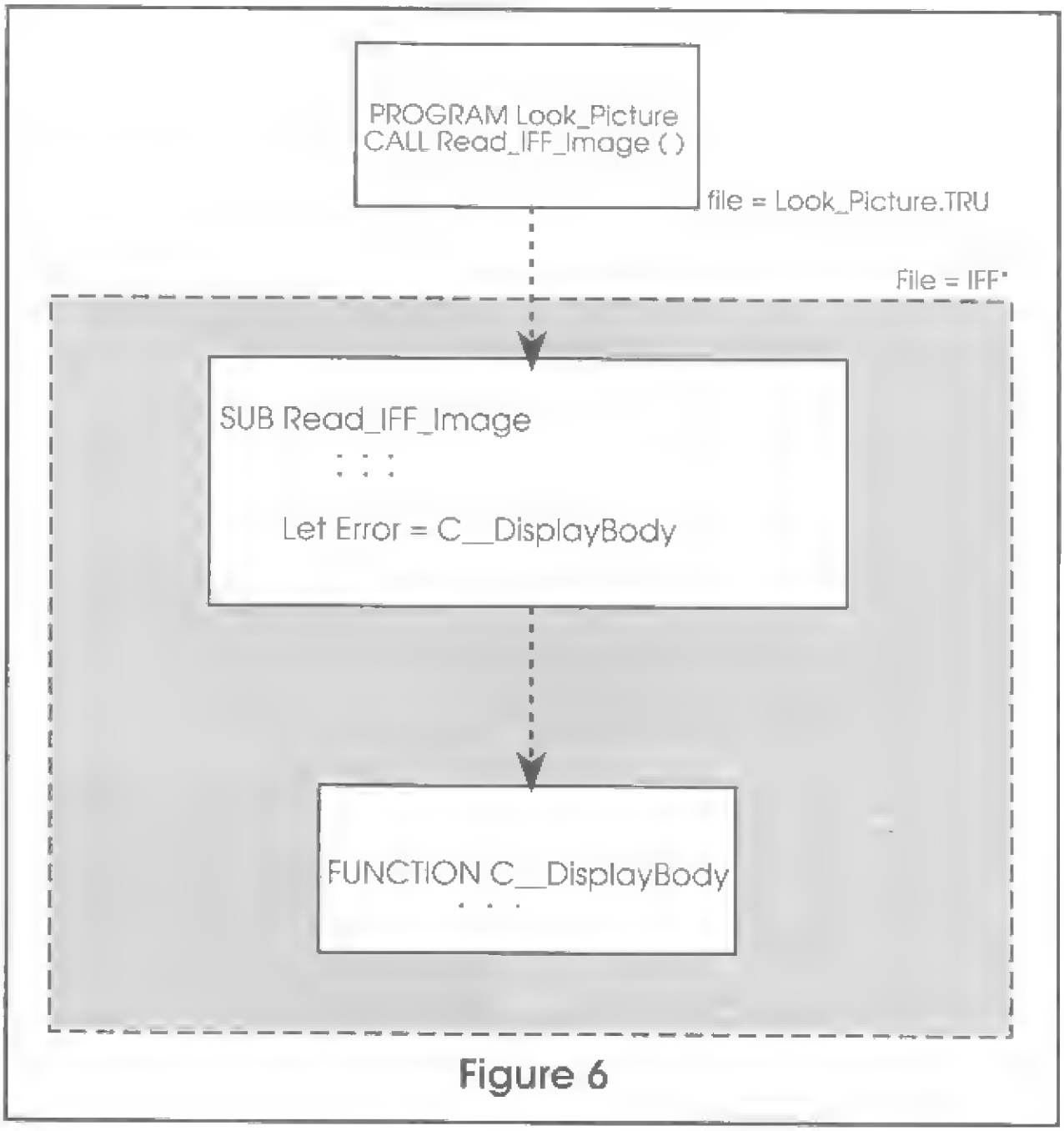


Figure 6

I have used True BASIC's EXCEPTION feature to protect the system in the event that the file "Gorilla.IFF" is a HAM or EHB image, as outlined in the related doc file on the True BASIC disk. Should a system error occur while in one of those modes, the exception handler, the part between the USE and END WHEN instructions, will get executed. Thus, the MODE_OFF SUBROUTINE will always get invoked. The entire section between the WHEN EXCEPTION and USE instructions is protected, including any SUBROUTINES or FUNCTIONS that get invoked by that section.

One last word about the use of this utility. It is advised that, once you have imported an image into True BASIC, you convert it to True BASIC's own format, which is called BOX KEEP. Doing so will not only allow True BASIC to read it from disk more quickly, but also to read it without displaying it immediately. Thus, a program that uses, say, five images, can load them all during the initialization phase of its operation, without the user even knowing about it, and keep them in memory as strings until they are required. True BASIC's BOX SHOW instruction can then be used to render them to screen, which, incidentally, executes so fast it can be used to display framed animations.

FILES ON THE MAGAZINE DISK

All files representing the design of this True BASIC language extension are on the magazine disk. IFF.TRU contains the True BASIC source for the library. The C___#? files contain the C routines that are used by the various routines within IFF.TRU. The files with an asterisk have been compiled and converted using the FinalTouch program. Note that within the IFF.TRU file there exist several LIBRARY instructions needed to access the various C routines. But for this to work all these files must exist in your current working directory. In contrast, the IFF* compiled LIBRARY contains a compiled version of IFF.TRU, as well as all the C routines, all in one convenient file. This was done using the MakeLib program, as I described above. It is intended that you place the IFF* LIBRARY file in your AmigaTools drawer and access it by writing the following instruction at the top of your programs:

```
LIBRARY "[AmigaTools:]IFF*"
```

Note that this particular design has been accepted by the company True BASIC, Inc., for inclusion into their latest "Student Edition" product for the Amiga, which you need to purchase in order to run it. Thus, you will get the latest version of IFF* when you purchase True BASIC.

So why, you ask, is the source code for what seems to be an official part of a language system being openly distributed on a magazine disk? Because both I and True BASIC, Inc., want it to be more than just a utility to use. We want it to be one to learn from as well. Hopefully, what you see here will convince you of the power of True BASIC on the Amiga and inspire you to try a few similar designs yourself. But even if you never do that, even if you simply use the utility to enhance your own True BASIC programs, it is still important that you have the source code. There is nothing more frustrating than trying to use a programming utility that does everything you want except for one minor detail, and having no way of doing anything about it. Having the source code gives you a chance to make it work the way you want it to. Feel free to improve the Read_IFF_Image routine to meet your own needs. If you make significant improvements, you might consider publishing your work so we can all profit.

The new "Student Edition" of True BASIC, version 2.0 for the Amiga, is available from:

True BASIC, Inc.
12 Commerce Ave
West Lebanon, NH 03784-9758
(800) 872-2742
(603) 298-8517
FAX: (603) 298-7015

Please write to:
Paul Castonguay
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

No Mousing Around

by Jeff Dickson

The Intuition pointer (Mouse) stays wherever you leave it and sometimes can obscure a portion of the screen you're looking at. While the actual size of it is small, the pointer is rendered at an angle which means, for example, that one or two characters of text can be hidden. This could be annoying when editing text files and perhaps more so during programming, because in some languages (e.g., Assembly), one or two characters can make a difference. Often, your only recourse under these circumstances is to physically move the mouse.

The Intuition pointer is implemented as a sprite, but, because sprite 0 is used, it cannot be covered up. Programs that open windows can change the appearance of the pointer, including hiding it, but few do. The only other option open to us is to move the pointer under software control.

The Input Device is used to accomplish this feat. An input handler is inserted ahead of Intuition into the Input Device "food chain." Of the many varied types of input events that can come our way, the only two of interest are IECLASS_TIMER and IECLASS_RAWMOUSE. Timer events are used to maintain an interval timer. Mouse input events are used as a cue to either restore the pointer's original position or to sample its current X/Y coordinates and restart the timer. Movement of the pointer is handled by the IECLASS_POINTERPOS input event. This input event is special in that Intuition treats the supplied pointer's coordinates therein to be relative to the origin of the Intuition view (upper lefthand corner of the screen).

In actual operation, the interval timer is decremented every time an IECLASS_TIMER input event is seen. If an IECLASS_RAWMOUSE input event occurs before the timer has expired, the pointer's current X/Y coordinates are sampled and the timer is restarted. If, however, the timer does expire, then the pointer is moved using an IECLASS_POINTERPOS input event to the upper-left corner of the screen. Anytime afterwards, if the mouse is physically disturbed in some manner (i.e., moved, button pressed), another IECLASS_POINTERPOS input event is inserted, but this time contains the pointer's last sampled coordinates so that Intuition will redisplay the pointer back where it originally rested. In both cases, the IECLASS_POINTERPOS input event precedes the input event that caused the action taken. For the latter, this could be especially important if, for instance, the pointer hovered above a gadget, was moved "out of the way," and then the user depressed the select button (left). At this point, the entire procedure starts over.

The IECLASS_RAWMOUSE input event includes an X/Y position field, but the coordinates of the pointer therein are relative to its previous position. For this reason, the pointer's true X/Y coordinates are obtained directly from the IntuitionBase MouseX and MouseY fields. Because these two fields are directly accessible, meaning that no pointers that could change need be dereferenced, IntuitionBase is not locked. To play it safe though, both fields are sampled at the same time. This prevents the possibility of an inaccurate reading.

The program is written in C and Assembler. C is used to initially install the input handler and tear it down when a control-C is sent to the process from which it was begun. The input handler, written in Assembler, performs all the grunt work necessary to move the mouse when it remains inactive for the preset time and restore it when the mouse is disturbed in some manner. Input handlers have the option of being passed a data area when they are invoked by the Input Device task, but for purposes of simplicity only the value of register A4 is passed. The input handler always presets register A4 to this value—meaning that variables declared in the C program are accessible by the input handler. This operation is synonymous with the GetA4() function provided by most compilers.

The program has been tested under both OS 1.3 and 2.04. It is recommended that a copy of the 1.3 ROM Kernel Manual be close at hand, because the intricasies of adding Input Device handlers, opening devices, etc., are not elaborated upon here. Also, it will help you to better understand the program. Enjoy!

Building the Program

The MANX C compiler package was used to assemble and compile the program. Below are the steps involved.

```
as InputHandler.asm
cc -o NoMousingAround.o NoMousingAround.c
ln -o NoMousingAround NoMousingAround.o
InputHandler.o -lc
```

The program accepts a single numeric argument. This gives you more control over how long the mouse must remain idle before the program moves it. Typically, every unit represents a tenth of a second, but can vary depending on system load. If a number isn't specified, then this value defaults to 50, yielding an approximate delay of five seconds.

Listing One: Input Device Handler

```
    NoHousingAround Input Device Handler

    Version 1.3, 28 February 1992

        \\ / /
       V  V

;
;          '  '  '  '   MarkEton Microcomputer Systems
;

  Symbols known to outside world

    PUBLIC  _test4
    PUBLIC  _Preset
    PUBLIC  _InputHandler

; Symbols defined elsewhere

    XREF    _maxticks
    XREF    _downcntr
    XREF    _state
    XREF    _coordinates
    XREF    _ios
    XREF    _ilb

; InputEvent structure

ie_Class            EQU 4
ie_Code             EQU 6
ie_XY               EQU 10

; Input Events of interest

IECLASS_RAWMOUSE    EQU 2
IECLASS_POINTERPOS  EQU 4
IECLASS_TIMER       EQU 6

; IntuitionBase offsets

MouseX              EQU 70
MouseY              EQU 68

; Simply return the value of register A4

_test4:         move.l a4,d0
                rts

; Preset the mouse current X/Y coordinate.

_Preset:        movea.l _ilb,a0
                move.l MouseY/a0),d0
                swap d0                 ; MouseX and MouseY
are backwards
                move.l d0,_coordinates
                rts

;
;
; Input Handler code
;
; If the mouse is physically moved, but had not been moved by
us prior -
; restart the interval timer and keep track of the pointer's
current XY
; coordinates. If, however, we had moved the mouse ourselves -
ately restore
; it to where it originally was.
;

_InputHandler:  movem.l d2-a2/a4,-(sp)
                movea.l a0,a2           ; hang onto
original event
                movea.l a1,a4           ; get ptr to registers
A4
                move.l _state,d2        ; load into the
state bool.

                ; we're only interested in timer and mouse
event . Anything
                ; else - ignore

DiscernEvent:   move.b ie_Class(a2),d
                cmp.b #IECLASS_TIMER,d
                bne.s NotATimer

                ; if a timer event, decrement the [illegible]er
and if it
                ; should go to zero and the state bool is reset
 tell
                ; intuition to move the mouse to the origin of
the int'n
                ; tick view

                tst.l d2
                bne.s NextEvent         ; can't bother if
wrong state

                move.l _downcntr,d0     ; get down counter
                sub.l d1,d1
                not.l d1
                neg.l d1
                sub.l d1,d0             ; decrement it
                move.l d0,_downcntr     ; update new value
                bne.s NextEvent
                lea _ios,a1             ; ready made
InputEvent structure
                move.l a2,(a1)          ; our event
precedes others
                move.b #IECLASS_POINTERPOS,ie_Class(a1)
                sub.l d0,d0
                move.l d0,ie_xy(a1)
                movea.l a1,a2           ; make event new
start
                not.l d2                ; change state
                bra.s Leave             ; can't reuse until
seen by lower
                                        ; priority
handlers!!!

                ; if instead a mouse event, put the state bool
 to set - tell
                ; intuition to immediately move the mouse back
to where it
                ; originally was

NotATimer:      cmp.b #IECLASS_RAWMOUSE,d0
                bne.s NextEvent
                tst.l d2
                beq.s IS
                lea _ios,a1             ; ready made
InputEvent structure
                move.l a2,(a1)          ; our event
precedes others
                move.b #IECLASS_POINTERPOS,ie_Class(a1)
                move.l _coordinates,d0
                move.l d0,ie_xy(a1)
                movea.l a1,a2           ; make event new
start
                not.l d2                ; change state
                move.l _maxticks,d1
                move.l d0,_downcntr     ; restart interval
timer
                bra.s Leave             ; can't reuse until
seen by lower
                                        ; priority
handlers!!!

                ; if however the state bool - reset, just copy
 out of
```

```
                    ;update mouse offset X
                    restart the interval timer

.5:                 move.l  _ilb,a1
                    move.l  MouseYlall,d1
                    swap d0                     ; MouseX and MouseY
are backwards!
                    move.l d1,_coordinates
                    move.l _maxticks,d0
                    move.l d0,_downcntr

                ; process the next event in the possibly linked
list
                    ; input event

NextEvent:          move.l (a0),d0              ; get next event?
                    move.l d0,a0
                    bne.s DiscernEvent

                    ; all done this time!

Leave:              move.l d2,_state            ; update possibly
new state
                    move.l a2,d0                ; return start of
event chain
                    movem.l (sp)+,d2/a2/a4
                    rts

end
```

Listing Two: MakeFile

```
CFLAGS=

NoMousingAround:  NoMousingAround.c InputHandler
                                                ln -lcma
NoMousingAround.o InputHandler.o -lc

NoMousingAround.o:          NoMousingAround.c
InputHandler.o:             InputHandler.asm
```

Listing Three: NoMousingAround.c

```
 *  NoMousingAround
 *
 *  Version 1.2, 28 February 1990
 *
 *  .  .  .  /  .
 *  . V . V . ___
 *  .   .   .
 *          .       Marketron Microcomputer Systems
 *
 *
#include <stdio.h>
#include <exec/types.h>
#include <devices/inputevent.h>
#include <devices/input.h>
#include <exec/interrupts.h>
#include <exec/ports.h>
#include <exec/memory.h>
#include <libraries/dos.h>
#include <time.sys.h>

 * defines and typedefs *

#define IOSIZE (sizeof(struct IOStdReq))
#define INTSIZ (sizeof(struct Interrupt))
#define MEMF_OPTIONS (MEMF_CLEAR)
```

```
typedef [illegible] *IOStdPtr;
typedef struct Interrupt *INTPTR;
typedef struct MsgPort *MPPTR;

 * variables used by the Input Handler *

long state;
long maxticks;
long downcntr;
long ilb;
long coordinates;
struct InputEvent iec;

 * local declarations *

long *retval();
void Press();
void InputHandler();

main(argc, argv) int argc; char *argv[];
{
    IOSPTR iosrb;
    MPPTR mp;
    INTPTR intstr;

    if ((ilb = (long)OpenLibrary("[illegible]ration.library", 0L)) {
        if (iosrb = (IOSPTR)AllocMem(IOSIZE, MEMF_OPTIONS)) {
            if ((OpenDevice("input.device", 0L, iosrb, 0L)) {
                if (intstr = (INTPTR)AllocMem(INTSIZ,
MEMF_OPTIONS)) {
                    if (mp = CreatePort(0L, 0L)) {
                        if (argc < 2 || !(maxticks =
atol(argv[1]))
                            maxticks = [illegible];
                        Printf("maxticks: %ld\n", maxticks);
                        Printf("...");
                        intstr->is_Code = (APTR)InputHandler;
                        intstr->is_Data = (APTR)&intstr;
                        intstr->is_Node.ln_Pri = 100;
                        iosrb->io_Data = (APTR)intstr;
                        iosrb->io_Command = IND_ADDHANDLER;
                        iosrb->io_Message.mn_ReplyPort = mp;
                        iosrb->io_Message.mn_Node.ln_Type =
NT_MESSAGE;
                        DoIO(iosrb);
                        GetMsg(mp);
                        if (iosrb->io_Error) {
                            Wait(SIGBREAKF_CTRL_C);
                            iosrb->io_Data = (APTR)intstr;
                            iosrb->io_Command = IND_REMHANDLER;
                            iosrb->io_Message.mn_ReplyPort
mp;
                            DoIO(iosrb);
                            GetMsg(mp);
                        }
                        DeletePort(mp);
                    }
                    FreeMem(intstr, INTSIZ);
                }
                CloseDevice(iosrb);
            }
            FreeMem(iosrb, IOSIZE);
        }
        CloseLibrary(ilb);
    }
}
```

Please write to:
Jeff Dickson
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

Tape Drives

by Paul Gittings

WARNING: The procedure outlined in this article will more than likely void the warranty on your computer. It is also possible to damage your computer and peripherals if the steps are not carried out correctly. If you are unsure about your abilities in this area arrange for a qualified technician to install the tape drive for you.

You can lose files and data on your hard disk due to mistakes, power surges, or programs which run amok. We all know that backing up our hard disks is an important step in preventing such losses, but how many of us perform these backups on a regular basis if at all? The idea of sitting in front of the computer for about an hour, or longer, feeding it floppy disks is enough to put most of us off; however, we soon regret this once we have a problem with the hard disk.

One solution is to backup the hard disk onto a tape. Tapes can hold lots of data and they are a plug and play operation; you stick the tape in, start the backup program and then there is nothing for you to do until the backup finishes. I did a quick survey of the market dhere in Australia; I looked at the advertisments in two Australian Amiga magazines, *The Australian Commodore and Amiga Review* and *The Professional Amiga User Magazine*. I was not able to find one company in Australia which was selling a tape backup solution for the Amiga; however, I think this has now changed and GVP is now selling such a system in Australia. Unable to find an Amiga-specific solution I turned to the IBM-PC and UNIX markets in the hope of coming up with a "roll your own" solution.

In the IBM-PC world there are tape drives which connect to the floppy disk drive controller. These tape drives are relatively inexpensive and the tapes they use can hold in the range 40MB-80MB. They would be a nice solution but no company has developed an interface to connect these drives to the Amiga.

Next I turned to the UNIX systems. The in-things in the UNIX backup world are "Exerbyte" (which uses Video-8 tapes) and "DAT" (which uses Digital Audio Tapes) backup systems. Both of these systems usually have SCSI interfaces and can store 1GB-5GB on a single tape! The downside is that these drives have a price tag to match; they usually cost over $AUS2000—a bit steep for my budget. However, these drives are relatively new to the market and as their acceptance increases they displace the older systems that the UNIX sites used to use, the SCSI cartridge tape drives.

A cartridge tape looks somewhat like an overgrown cassette tape. The physical size of a cartridge tape is 10cm x 15cm x 1.5cm. The two common cartridge tape capacities are 60MB and 150MB. This size tape should cover most users needs, but what about the price?

The drives cost around $AUS1000 new, which is still steep for the average user. However, these drives, thanks to the "Exerbyte" and "DAT" drives, are starting to turn up on the second-hand market. I recently purchased a used Archive 2150S SCSI cartridge tape drive for $AUS295.

For Australian readers, I purchased the drive from Unlimited Computer Supplies, 60 Hawksview St, Guildford,

● installation
● formatting
● general setup

NSW; phone (2) 892-2775. The drive had been tested by the people at UCS and came with a three-month warranty. UCS still had another 28 or so of the 150MB Archive 2150S drives left as well as a number of 60MB tape drives (for under $AUS200), and a number of 150MB cartridge tapes for $AUS50. All items were second hand. If UCS has sold out, check other second hand computer stores or even the computer equipment classified advertisments in your local newspaper.

The Archive 2150S is classified as a 150MB/250MB tape drive and can read/write the following cartridge tapes:

DC6150 (150MB)
DC6250 (250MB)
DC6320 (150MB)
DC6525 (250MB)

The 2150S will also read the following tapes as well:

DC300
DC300XL
DC300XLP
DC450
DC450XL
DC600
DC600A

The drive will actually write to the DC600/DC600A tapes and will store about 120MB, but you should remember that the DC600/DC600A tapes are certified for use in 9-track tape drives and the 2150S is an 18-track tape drive. You should also be aware that the DC6250 tapes cause the tape head to wear in such a way that eventually you may not be able to read/write the other formats. As a general rule of thumb, stick to one format. The price for tapes themselves start at about $AUS30 and go up from there.

I had the drive, and now the problem was how to hook it up to my system. My system consists of an Amiga 2000, with a GVP Series II SCSI controller and a Seagate hard drive. The hard drive is mounted on the controller card, meaning that the large 5-inch bay in the 2000 was free; this is where I decided to put the tape drive.

If you have already filled up the 5.25-inch bay on your A2000 or you have an A500, A600, A1000, or A3000, you will need to get an external case for the tape drive. The external case will have to have its own power supply. The power requirements for a tape drive are rather high. The power specifications marked on 2150S tape drive that I bought were:

+5 VDC, 17.5 Watts
+12 VDC, 48.0 Watts

Using what I remember of my high school electronics this works out as:

3.5 Amps at +5
4 Amps at +12

A power supply rated at about 65 Watts and designed to drive a 5.25-inch hard drive should do the job. NOTE: Phoenix Microtechnologies in South Australia advertises a "Phoenix SCSI Box"; you may wish to contact them to see if this box is suitable for use with a tape drive.

Before installing the tape drive into the 2000 (or expansion box) there is some preparation work to be done first. You will of course need some sort of cable to hook up your drive to the controller. The type of cable will depend on your setup. If the drive is to be mounted in an external box, you will probably need a 25- to 50-pin shielded SCSI cable.

If you are mounting the tape drive internal to your computer, then you will need to obtain a flat 50 conductor cable with 50-pin IDC sockets on it. The number of sockets required and their location will depend on the number of drives in your computer (0,1,2) and their location. The cable and the connectors can be obtained from your local electronics store.

When attaching the connectors to the cable, use a vice to close the connectors onto the cable; I tried the hammer method and broke two connectors before giving up and getting a very competent technician to make the cable for me. If you do try to build your own cable, remember that the key on all the sockets should point in the same direction and leave extra room between the sockets just in case you need to reroute the cable later. Make sure that when you install the cable that the controller is at one end of the cable.

Next we need to prepare the tape drive itself. This will involve the setting of several jumpers on the back of the tape drive. The jumpers differ from drive to drive so when you

A Great Reason to Own an Amiga

Amazing Computing provides its readers with:

- In-depth reviews and tutorials
- Informative columns
- Latest announcements as soon as they are released
- Worldwide Amiga Trade Show coverage
- Programming Tips and tutorials
- Hardware Projects
- The latest non-commercial software

Order a SuperSub and get this great Amiga peripheral

AC's GUIDE is recognized as the world's best authority on Amiga products and services. Amiga Dealers swear by this volume as their bible for Amiga information. With complete listings of every software product, hardware product, service, vendor, and even user groups, *AC's GUIDE* is the one source for everything in the Amiga market.

AC's GUIDE also includes a directory of Freely Redistributable Software from the Fred Fish Collection and others. For Commodore executives, Amiga dealers, Amiga developers, and Amiga users everywhere, there is no better reference for the Commodore Amiga than *AC's GUIDE to the Commodore Amiga.*

12 Issues of Amazing + 2 AC's GUIDEs!

A Great Reason to Get Into Your Amiga

AC's TECH offers these great benefits:

- The only disk-based Amiga technical magazine
- Hardware projects
- Software tutorials
- Interesting and insightful techniques and programs
- Complete listings on disk
- Amiga beginner and developer topics

Call NOW for more information about these great Amiga accessories from Amazing Computing!

1-800-345-3360

buy your tape drive, try to get some documentation on the jumper settings for it. This article will use the jumper settings of an Archive 2150S tape drive as an example.

With the tape drive right way up, turn it around so the back end is facing you.

The jumper block is what we are currently interested in. A jumper is slid over two adjacent pins on the jumper block to select/enable an option or choice. When you buy your tape drive see if you can get several of these jumpers; the ones with tabs on if possible since they are easier to remove. An alternative would be to take along this article and ask, very nicely, if they could set up the tape drive options for you. You should also be able to get jumpers at your local electronics stores

The jumper block on the 2150S is arranged as follows (same orientation as above) where each "[ ]" represents two pins:

```
[ SERIAL ]      [ CF2  ]      [ ID2  ]
[ DIAG.  ]      [ CF1  ]      [ ID1  ]
[ PARITY ]      [ CF0  ]      [ ID0  ]
```

SERIAL - NOT USED. Do not jumper
DIAG - NOT USED. Do not jumper
PARITY - To enable parity you should place a jumper here. Whether or not this should be set depends on your controller; so check your documentation. Either every device connected to the controller has PARITY enabled or none of them do. In my setup no jumper appears here.
CF0-3 - Selects the number of bytes to be transferred over the bus. You may need to change this, explained below.
ID0-3 - Selects the SCSI ID number of the tape drive; you will more than likely need to change this.

First we need to set the SCSI ID number of the tape drive. The important thing here is to ensure that the ID number you set on the tape drive does not conflict with the ID of a device already connected to the SCSI controller. Most SCSI controllers use the ID number 7 for themselves. If you have a hard disk mounted on the controller card, the disk will usually use ID number 0. So you are left with a choice of 1-6.

Unit number 4 is the usual number used for a tape drive in Amiga systems; however, the choice is yours. On my system the tape drive unit number is 1. At the time I selected the number I did not have a complete table of the tape drive jumper settings and I took a guess at which of the ID jumpers would select ID number 4; I had a 50 percent chance of being right but I still got it wrong! So that you will not have the same problem here is a table of jumper settings for the SCSI IDs 0-6 for the Archive 2150S tape drive:

```
ID2: o o   o o   o o   o o  [o=o] [o=o] [o=o]
ID1: o o   o o  [o=o] [o=o]  o o   o o  [o=o]
ID0: o o  [o=o] o o  [o=o]   o o  [o=o]  o o
SCSI ID:    0     1     2     3     4     5     6
```

Where "[o=o]" indicates that a jumper should be installed.

The next option to set is the buffer size. This sets the number of bytes to be transferred over the bus in a single operation. When I first installed the drive, the size was set at 2K and this seemed to work. However, in one of Thad Floryan's Usenet News articles in the "comp.peripherals.scsi" newsgroup he points out that the SCSI "copy" command requires a minimum of 16K and he recommends that the buffer size be set to 16K. I have followed Mr. Floryan's advice and have since set the buffer size to 16K. Here are the jumper settings for the various buffer sizes for an Archive 2150S tape drive:

```
CF2:  o o   o o   o o   o o  [o=o] [o=o] [o=o] [o=o]
CF1:  o o   o o  [o=o] [o=o]  o o   o o  [o=o] [o=o]
CF0:  o o  [o=o]  o o  [o=o]  o o  [o=o]  o o  [o=o]
SIZE: 2K    4K    6K    8K    12K   16K   24K   32K
```

Now that the jumper block is configured we move to the terminators. No, not the killer robots but the resistors that have to be installed in the last device in the chain of SCSI devices. The purpose of the terminating resistors is to stop "cross-talk," "reflections," and other electrical problems that can occur when electronic signals are sent over long stretches of cable.

On the Archive 2150S there are three terminators located in a row just above the SCSI connector. The terminator packs look a little like very small squashed sausages with eight pins sticking out the bottom (If you have a "GVP Series II SCSI/RAM Controller Card User's Guide" there is a drawing of a terminator pack on page 7). On tape drives other than the Archive 2150S the terminators may not be in the normal packs but may come in packages (like ICs) or connectors. Again, check the documentation which came with your tape drive before you start pulling bits and pieces off it. When you purchase your tape drive, try to get one with the terminators already installed. While they are easy to remove, they can sometimes be a pain to insert.

As stated earlier the terminator packs should be installed on and only on the last device in the chain. Getting it wrong can cause all sorts of intermittent problems, so if your system doesn't seem to work properly after installing a SCSI device, double check the location of the terminating resistors.

As the tape drive is the last device in the chain, the terminators will be installed on the tape drive and only on the tape drive. If the hard disk had terminator packs installed they would have had to be removed once the tape drive was added. If you have more than one internal hard drive just make sure that whichever device is at the end of the chain has terminator packs installed and all other devices have their terminators removed. *Note: a disk drive mounted on the controller card will usually have no terminators; check the documentation that came with your controller card and the hard disk.*

If you have external drives, then the terminators are removed from all internal devices and installed in the last device in the external chain I know this is true for the GVP series II controller. If you have a different controller, check the documentation that came with it. So if your tape drive is not the last device in the external chain or it is an internal drive, and you have external devices, then remove the terminating

resistors from the tape drive.

Once the terminator issue is taken care of we are ready to physically install the tape drive. Before starting, turn off the power to your Amiga and remove all connectors and move the machine to a large, flat, and well-lit work area. To install the drive in the Amiga 2000's 5.25-inch bay, you first need to remove the cover which is held on by five screws. Put the screws somewhere safe. Once the cover is off you have two options:

1) Remove the entire drive bay chasis.
2) Attempt to install the drive while the chasis is in place.

I chose option 2) but I have been told that 1) is easier as it makes the mounting screw holes more accessible.

Since I chose option 2), I had to remove the SCSI controller card to get at mounting holes on the drive bay; if you have other cards in your 2000, you may have to remove them as well. Once I had the controller out I removed the old SCSI cable from the controller card and my hard disk drive.

I next attached one end of the new SCSI cable to my tape drive and fed the cable through the drive bay. It's easier to connect the cable to the tape drive before it is mounted in the bay. The SCSI connector on my tape drive was a bit damaged and I had difficulty plugging in the SCSI cable. It turned out that some of the pins had been slightly bent and no longer lined up properly with the holes in the socket; I used a screw driver, very gently, to bend the pins back into alignment. A wiser strategy would have been to return the drive for a replacement but I am a tad impatient.

Next I slid the drive into the bay. The tape drive has four screw holes which have to be aligned with four slots in the drive bay. You may need to get somebody to hold the drive while you drive in the screws since the tape drive does not rest on the bottom of the bay. I obtained the mounting screws from of my junk box, but you may want to ask for the screws when you buy the tape drive. Don't tighten the screws all the way up, since you will have to align the front of the drive with the front of the floppy disk drive. By having the screws loose, you can mpve the tape drive backwards and forwards a little bit in the bay, allowing you to align the front of the tape drive with the front of the floppy disk drive. This will then ensure that the front of the tape drive will be flush with the cover when it is replaced. Once you have the front of the tape drive aligned with the front of the floppy disk drive, tighten up all four screws.

Connecting the tape drive to the power supply is simply a matter of taking one of the four-pin power connecters inside tha Amiga 2000 and plugging into the power socket on the back of the tape drive. The power connectors are keyed so they will go in only one way.

I then connected the SCSI cable to my hard drive and controller and reinstalled the controller card. Before putting the cover back on the 2000, I removed the cut out for the 5.25-inch bay in the front of the cover; this is held in place by two screws. I put the cover back on and plugged in all the connectors. Then came the first big test ... I turned the power on.

The light on the front of the tape drive came on for a short period of time, and the machine booted normally, a big sigh of relief. So the tape drive was now physically installed but I still had to configure the system and install some software before I could make any use of it.

Depending on how you previously configured your system, you may have to make changes to your hard disk configuration so that the SCSI device driver software will be able to locate and identify the new tape drive. To check the configuration of your SCSI devices, use the hard disk preparation program that was supplied with your controller card. This program should be able to display information on all the devices connected to the controller. If the program does not show that there is a device at the ID number you allocated to the tape drive, then you will probably need to make some changes to the configuration of the hard drives; it could also mean you have cabling, termination, or an ID selection problem but check the hard disk configuration first.

To check on my hard disk configuration, I used the hard disk preparation program supplied with the GVP controller, *FaaastPrep*, and I went to the "Manual Installation" screen. Initially this screen shows the device at ID number 0; in my case my one and only hard disk. I then clicked on the ">" gadget to increment the ID number and I cycled through all the ID numbers but no other devices were shown. Obviously something was not correct. I clicked on the "<" gadget to decrement the ID number back to 0 to double check the configuration of my hard drive and there I noticed a possible cause of my problem; the "Last LUN" (LUN = Last Unit Number) and "Last Disk" gadgets were selected. These two options when selected inform the SCSI device driver that the device at this ID number is the highest device number connected to the controller. In my case, this meant that the SCSI software would not look for any SCSI devices with an ID number higher than 0, so the software could not locate the tape drive which has an ID of 1. To remedy this, I deselected the "Last LUN" and "Last Disk" option for device number 0 and wrote back the changes. Then I rebooted my machine.

This time it took longer for my system to come up. The reason for this delay is that when you boot your system the SCSI controller goes out and polls all the device ID numbers from 0-6 to see which devices are connected. It will wait a certain amount of time before it decides there is no device at any given number. Previously my system would only poll device number 0 (since this device number had been selected as being the "Last LUN" and "last Disk"). Now, however, the controller polled all device numbers from 0-6 and this takes longer (unfortunately you can't make a tape drive the "Last LUN" and "Last Disk").

Once the machine was rebooted, I needed to check to see if the tape drive was now shown as being connected to the controller. Rather than use FaaastPrep again, I thought I would try out the hard disk preparation program supplied with AmigaDOS 2.0, HDToolBox.

To use HDtoolBox with non-Commodore SCSI controllers, you have to inform HDToolBox of the name of the device driver that is used with your SCSI controller. To do this, you set the HDToolBox icon tool type "SCSI_DEVICE_NAME" equal to the name of the device driver. For instance, to get HDToolBox to work with a GVP controller, select the "HDToolBox" icon, select "Information" from the WorkBench

2.0 "Icon" menu. When the "Information" window is displayed, click on the "New" button and enter

```
SCSI_DEVICE_NAME=gvpscsi.device
```

Then click on save.

Well I double clicked on the HDToolBox icon and up came a list of ID numbers and associated drive types. My hard drive was listed but there was no sign of the tape drive, all the other ID numbers showed "unknown" as the drive type. Now to save you the trouble that I went through rechecking every thing and having a minor anxiety attack, I will quote a line from the "HDToolBox" section of the AmigaDOS 2.4 manual, page 6-53:

"SCSI tape drives are always listed in Drive Type as unknown"

However, I have heard from a person with a Commodore SCSI controller that his connected tape drive is listed when using HDToolBox; so the problem may be with the GVP controller and/or software.

I resorted to using "FaaastPrep" again. I went to the "Manual Installation" screen and cliked on the ">" gadget to increase the ID number and sure enough there at ID number 1 was an entry for the tape drive. It showed the name as "ARCHIVE" and the manufacturer's ID as "VIPER 150 21731".

The next step was to try to write, then read, data from the tape. To do this, I decided to use the hard disk backup program supplied with AmigaDOS 2.0, "HDBackup". This turned out to be a mistake since "HDBackup" and a related program "BRU" seem not to work with tape drives which are connected to a non-Commodore SCSI controller; the tapes appear to get written OK but any attempt to read the tape results in a complete lock up of the computer. I have sent electronic mail to Commodore and a fax to GVP describing the problem. GVP replied to say that the problem is with "BRU" and that a new version will be released. In case you have a Commodore SCSI controller (A2090, A2091, A590, A3000) I am including the steps required to configure "HDBackup".

"HDBackup" is in fact a graphical front end to another program supplied with AmigaDOS 2.0 called "BRU". "BRU" is a hard disk backup program that is run from the Shell/CLI. To get "HDBackup" to work, you therefore need to configure "BRU" as well as "HDBackup".

To configure "BRU," you have to edit the "BRU" configuration file "s:BRUtab". Find the entry in this file which starts with "tape:". Then replace the field "device = scsi.device" with the name of the device driver supplied with your SCSI controller. If you have a Commodore controller you don't need to change this. GVP users would use "device = gvpscsi.device". If the ID number you selected for your tape drive was a number other than 4, then you also need to change the "unit = 4" entry. On my system the tape drive ID is currently 1 so the BRUtab entry had to be changed to "unit = 1". You can also do the same thing for the "ntape:" entry.

"ntape:" is a non-rewinding version of "tape:". If you were to select "ntape:" as the device to which "BRU/HDBackup" writes the backup archive to, the tape drive would not rewind to the beginning of the tape before writing out the archive. On the other hand, had you selected "tape:" as the output device, then the tape drive would first rewind the tape to the beginning before writing out the archive.

The device specified by the first entry in the "s:BRUtab" is used as the default output device by "BRU". So, if you want the tape device to be the default backup device, move the entire "tape:" entry to the top of the "s:BRUtab" file.

To configure HDBackup, you need to add the following tool type to the HDBackup icon:

```
DEVS   tape: | ntape:
```

If you want "HDBackup" to use the tape drive as the default backup device add the following tool type after the "DEVS" tool type:

```
USE = tape:
```

Save these changes. "HDBackup/BRU" should now be configured, and if you don't have a GVP controller, they may even work!

Having a tape drive is not much use without having a way of writing/reading tapes, so I started to look at my options. The first was to buy a commercial backup program. My next option was to check the wonderful world of public domain software to see if there was a solution available there. After scanning the Fred Fish catalogue I found a program called "BTNtape" or the "Better Than Nothing" tape handler. This program is a device driver for a tape drive. It allows you to access a tape drive in a similar fashion as you would other devices. There are a number of versions of this program around and the earlier versions do have problems, so make sure you get at least version 2.1 (Fish Disk 558).

To install the tape driver, copy the file "btn-handler" to your "L:" directory and then edit the file "devs:MountList" and add the following entry:

```
TAPE: Handler = L:btn-handler
Priority = 5
Stacksize = 4000
GlobVec = -1  Startup
 "gvpscsi.device/UN-1/BT-4/NB-32" ■
```

If you want the "tape:" device mounted automatically when you boot your system insert the line:

```
Mount = 1
```

in the above. You may need to change the "Startup" line. The "gvpscsi.device" should be the name of the device driver supplied by the maker of your controller card; i.e. Commodore controller users would use "scsi.device." The "UN-" entry specifies the ID number of the tape drive. If your tape drive ID is set 4, then you would use "UN-4." If you have no FastRAM (or you are not sure) then you need to set the buffer memory type option to 2 (BT-2) to select Chip RAM. To mount the tape device use the command:

```
mount tape:
```

Now put a tape in the drive and try to copy a file on to the tape with:

```
copy filename TO tape:
```

The tape should make some noise and the light should come on; it may have to rewind the tape first so be patient. Once this has been completed, change to another directory and enter:

```
copy tape: TO filename
```

The tape should again make some noise and eventually you should be left with a copy of the original file in the current directory. If you have any problems, double check the MountList file, fix it if there are problems, and remove the incorrect device with:

```
assign tape: REMOVE
```

before mounting the fixed version with:

```
mount tape:
```

If you still have problems, check the documentation that came with BTNtape to see if there is anything special that has to be done for BTNtape to work with your SCSI controller.

The "tape:" device is a streaming device which means that it doesn't support a filesystem, i.e., it doesn't support directories or even file names. So you cannot copy to/from "tape:filename". You can, however, modify the behavior of the "tape:" device by refering to the "tape:" device with keywords attached to the right.

tape: will rewind the tape to the beginning and starts the read/write operation from there. tape:NR starts read/write operations at the current tape position. tape:APP goes to the end of the last written file and starts the write operations there. Think of this as an append operation. It does not work with all drives. tape:num starts a read operation at the file specified by the number. The first file on the tape is number 0.

While these commands would allow you to read and write any number of files to the tape, it would be a major task trying to keep track of which files are located at which positions on the tape. It would not be a very good solution for a backup system by themselves. There's a better way.

There is a program called "tar" (Tape Archiver) which was originally written on UNIX machines to handle exactly this situation. "tar" will back up any number of files including entire directory trees to a tape in an archive file. Using "tar," you can access the files in the archive by their names so you no longer need to know where the file is located on the tape. The author of "BTNtape," Rob Rethemeyer, recommends the version of "tar" ported by Jonathan Hue which can be found on Fish Disk 445. "tar" commands have the form:

```
tar [flags] [archive_name] [file/directory names]
```

where the [flags] are a group of letters preceeded by a "-" which specify the type of operation to be applied to the archive file, [archive_name].

The most used flags are:
c - create the archive [archive_name] and add all the files and directories specified on the command line to it.

f - Tells tar to use the next name as the name of the archive. This archive_name could be a file name if you want to write the archive file to a disk. If you want the archive file to be placed on the tape you should use "tape:" as the archive_name.

t - lists the contents of the tar file.

v - verbose mode, tar prints messages to explain what it is doing.

x - extract the files from the archive.

For example, to back up a directory structure, you could use the following command;

```
tar -cvf tape: dir_name
```

where dir_name is the name of the directory you want to back up. "tar" if given a directory name will do a "recursive" backup - that is it will copy all the files in the specified directory and the directories below it to the archive file. If you want to see what is in the archive file:

```
tar -tvf tape:
```

will list the table of contents of the archive. To restore the directory structure from the tape use:

```
tar -xvf tape:
```

If you just want to restore a single file use:

```
tar -xvf tape: path/name
```

To archive an entire partition, say HD0:, change directory to HD0: and enter:

```
tar -xvf tape: ""
```

This will write the entire partition to the tape while you can go off and enjoy a Coke.

"tar" has many more options than are discussed here; to find out more read the "tar.man" file that comes with the "tar" program on Fish disk 445. The "tar" program on Fish disk 445 is compatible with the "tar" program in use on many computers running the UNIX operating system. Using "BTNtape" and "tar," I have been able to backup up entire directory structures on my Amiga, then take the tape to work and read the directory structure off the tape using the "tar" program and tape drive on a Sun SPARCstation; the reverse works too.

The documentation which comes with BTNtape has information on the use of "tar" as well as the "btn-handler." It also includes many helpful hints on how to manage your backup tapes. These include a reminder to always label your

tapes with their contents and the date the tape was written. Always do a " tar -tvf tape:" command after writing out a tar archive, since this will check the integrity of the tar archive as well as listing the contents of the archive. Make up a floppy boot disk which has the "btn-handler" installed on it, then in the case you cannot boot off your hard disk, you will be able to boot off this floppy and restore the hard disk from your backup tapes. Of course you have to make the backups for this to work.

How often you backup your system will depend on how quickly things change in your system and how you partitioned your hard disk. For instance, I have a work partition which changes almost daily while my system partition goes for weeks, even months, without major change. To handle daily updates, I still use a floppy. I keep a floppy disk in df0:, to stop the drive from clicking, and at the end of a "session" I copy the directories I have been working on to the floppy disk. Now that my tape drive is set up I am considering doing a complete backup of my system (all partitions) once a month and a weekly backup of the work partition.

So that is it; with the addition of a SCSI cartridge tape drive, "BTNtape," and "tar," I have a workable backup solution which, while not perfect, at least takes much of the drudgery out of the process. "tar" is not the most user friendly of backup programs, but it does work and is reliable and is very useful if you need to exchange data with UNIX systems. If you would like a program with a graphical interface you should look at a commercial product. Or you could wait awhile because I plan, when I get the time, on writing some ARexx scripts to automate and manage my backups for me. So keep your eyes open for a future article and in the meantime, back up those hard disks.

If you have access to ACSnet, AARnet, or Internet you can contact me by electronic mail, my address is "paulg@tplrd.tpl.oz.au".

BIBLIOGRAPHY

"BTNtape.doc" by Rob Rethemeyer, 1991. Can be found on Fish disk number 558.
"GVP SERIES II SCSI/RAM Controller Cards; User's Guide" by Great Valley Products, Inc. 1990
Thad Floryan's periodic postings to the Usenet News in the "comp.peripherals.scsi" newsgroup.
"SystemSoftware; AmigaDOS 2.0 manual" by Commodore.
"tar.man" by Jonathan Hue, can be found on Fish disk 445.

Please write to:
Paul Gittings
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

A Tape Drive Utility

Included with the BTNtape software on Fish Disk 558 is a small utility program which aids handling of tapes. The program is written in ARexx and is called "tape.rexx". This program allows you to rewind, retension, and erase tapes. To install the program, simply copy it from the fish disk to your REXX: directory. To run "tape.rexx," enter the following:

```
rx tape [cmd [device] ]
```

where:

device - is the name of the tape device as specified in your mountlist but without the ":". The default "tape" will be used if no name is supplied.

cmd - can be one of the following;

BACK - skip back one file.
END - goto the end of data on this tape.
ERASE - erase the entire contents of the tape.
FORWARD - skip forward one file.
RETENSION - this will retension the tape by going to the end of the tape and rewinding to the beginning.
REWIND - will rewind the tape to the beginning.
WFM - At the end of each file (or tar archive) a file mark is written to the tape. This command will allow you to write an additional file mark for the systems that need them.

For example, to rewind the tape in the drive mounted as "rst0:" you would use the following command:

```
rx tape rewind rst0
```

—Paul Gittings

Entropy in Coding Theory

by Joseph J. Graf

Many people believe the goal of data compression to be the removal of redundant data. This belief is not entirely true. The goal of data compression is to express data more efficiently without losing information, which has the effect of removing redundant information.

The Huffman method of data compression uses the frequency of information as a basis of symbol generation; the more frequent the value, the shorter the symbol size. For example, the letter A is more frequently seen in text than Q so A would use fewer bits to represent than the letter Q.

The Huffman model of data compression is based on the coding theory of Shannon. You need not know the full implementation of the Huffman scheme to determine the compressibility of a file. Shannon's entropy formula calculates that for you.

Entropy is defined by Webster's New Collegiate Dictionary as "a measure of the amount of information in a message that is based on the logarithm of the number of possible equivalent messages." The formula to determine the actual amount of information contained in a message is -(p * log2 p), where p is the frequency of a character in a file. Since we are working in binary instead of base 10, a logarithm function that works in base two is necessary. From calculus, the derivation of a base n logarithm function is as follows: logn x = log10 x / log10 n. This is the same as logn x = log10 x * (1 / log10 n). We want to do as few computations as possible for speed, so a little calculator work yields the value of 3.219 for (1 / log10 2). The new formula is log2 x = log10 x * 3.3219.

The entropy function is in terms of x, where x is the frequency of any given value. To determine the frequency a value has in a file, divide the number of occurrences of the value by the size of the file in bytes. Use this value for x in our new log function and you have the entropy for that particular value. A tally must be kept of all values' entropy, which in the end will yield the compressibility of the file.

The first program (Listing 1) calculates the savings potential using an order-0 compression scheme. An order-0 scheme looks only at one value at a time, an order-1 scheme looks one place ahead, order-2 looks two places ahead and so on. The higher order methods require more processing and more memory to perform compression, although the compression ratios grow larger.

The second program (Listing 2) calculates the entropy of a file for an order-2 method. By noticing the small differences between the programs, the code can be adapted to determine the compressibility of a file for an order-n scheme.

Order 0

```
/******* ORDER0 by Joseph J Graf *******

#include <stdio.h>
#include <math.h>

#define MAX 256

FILE *input;
unsigned int table[MAX];
long size;


void file_read(name)
char *name;
{
char ch;

        if(! (input = topen(name,"r"))
        {
                printf("\n Couldn't open file:
```

```
%s!:\n",name);
                exit(0);
        }

        while( (ch = getc(input)) != EOF)
        {
                table[ch]++;
        }

        size = ftell(input);
        fclose(input);

        return;
}


double calc_entropy()
{
double entropy = 0.0;
double freq;
register int i;

        for(i = 0; i < MAX; i++)
        {
                if( table[i] )
                {
                        freq = (double)table[i]
/size;
                        entropy += -((freq *
(log10(freq) * 3.3219));
                }
        }

        return(entropy);
}



main(argc,argv)
int argc;
char *argv[];
{
double entropy, perc;

        if( (argc != 2) || (argc > 2) )
        {
                printf("\nUsage: order
filename\n\n");
                exit(0);
        }

        printf("\n\nReading...\n");

        file_read(argv[1]);

        printf("Determining ratios...\n");

        entropy = calc_entropy();
        perc = (100 - ((entropy * 100) / 8));

        printf("\n%s has an order-0 entropy
",argv[1]);
        printf("of %2.3f bits per byte.",entropy);
        printf("\n\nHuffman encoding would
shrink");
        printf(" %s %2.1f%%.\n\n",argv[1],perc);

        return(0);
}
```

Order 1

```
 ******* ORDER1 by Joseph J Graf *******/

#include <stdio.h>
#include <math.h>

#define MAX 256

FILE *input;
unsigned int table[MAX][MAX];
long size;


void file_read(name)
char *name;
{
char ch, ch2;

        if(( (input = fopen(name,"r")) )
        {
                printf("\n Couldn't open file:
%s!!\n",name);
                exit(0);
        }

        while( (ch = getc(input)) && (ch2 =
getc(input)) != EOF)
        {
                table[ch][ch2]++;
        }

        size = ftell(input);
        fclose(input);

        return;
}


double calc_entropy()
```

```
{
double entropy = 0.0;
double freq;
register int i,j;

      for(i = 0; i < MAX; i++)
      {
            for(j = 0; j < MAX; j++)
            {
                  if( table[i][j] )
                  {
                        freq =
(double)table[i][j] / size;
                        entropy += -(freq
* (log10(freq) * 3.3219));
                  }
            }
      }

      return(entropy);
}



main(argc,argv)
int argc;
char *argv[];
{
double entropy, perc;

      if( (argc == 1) || (argc > 2) )
      {
            printf("\nUsage: order1
filename\n\n");
            exit(0);
      }

      printf("\n\nReading...\n");

      file_read(argv[1]);

      printf("Determining ratios...\n");

      entropy = calc_entropy();
      perc = (100 - ((entropy * 100) /
8));

      printf("\n%s has an order-1 entropy
",argv[1]);
      printf("of %2.3f bits per
byte.",entropy);
      printf("\n\nHuffman encoding would
```



```
shrink");
      printf(" %s
%2.1f%%.\n\n",argv[1],perc);

      return(0);
}
```

A Fast Plot Library

by Michael Griebling

Introduction

For some time now I have been searching for a general-purpose plotting library that is both easy to use and offers fairly sophisticated functionality which can produce professional-looking graphical plots. Commercial packages are generally too specific (i.e., business graphics only) to be used in a wide range of applications and also are fairly pricy. AREXX plotting libraries would be another alternative but they are interpreted and thus are considerably slower than a custom software package. In addition, the plotting capability should be easy to integrate into other programs with very simple calls and a minimum of set up. The PlotLibrary package meets these goals with the following features:

produces bar charts, linear, curved, scatter, logarithmic and semi-log plots
fast due to the use of a compiled language (Modula-2)
any number of plots on a custom screen with up to 32 colours
automatic x- and y-axis labelling or user-definable labels
automatic grids
automatic local minima/maxima labelling
information box
user-definable text/line colours

Some sample plots produced by the PlotLibrary routines are shown in Figures 1 to 5. The program in Listing 1 produced these different plots. Listing 2 is the complete source for the PlotLibrary module. Listing 3 shows the source for a support module called StringUtils.

The remainder of this article demonstrates how to create your own plots by walking through the program in Listing 1 and describing the PlotLibrary routines, as needed, to give a basic understanding of how to use them.

Creating Your Own Plots

The first step is to call the InitPlot plot routine with a set of arguments which define the plot characteristics as follows.

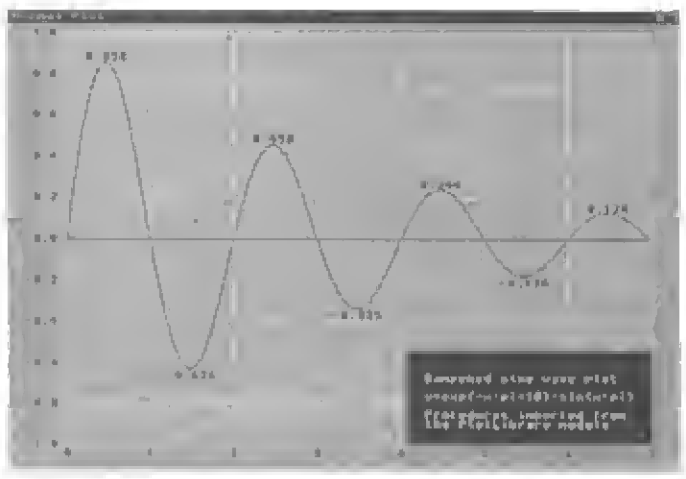


Figure One.

```
PROCEDURE                    InitPlot(
VAR Plot                     : PlotType;
MainTitle                    : ARRAY OF CHAR;
PlotKind                     :PlotKindType;
xMin, xMax, yMin, yMax : REAL;
width, height                : INTEGER;
xDiv, yDiv                   : CARDINAL;
xSubDiv, ySubDiv             : CARDINAL;
xDec, yDec                   : CARDINAL;
NumberOfColours              : CARDINAL)
        : BOOLEAN;
```

where the Plot is a variable which is used as a handle to identify a specific plot; MainTitle is the title which appears at the top of the plot; PlotKind is one of

1) normal (Figure 1)plot continuous curves,
2) line (Figure 2)plot points connected with line segments
3) bar (Figure 3)bars of various heights,
4) log x (not shown)plot the log of the x-axis;
5) log y (not shown)plot the log of the y-axis;
6) log-log (Figure 4)plot the log of the x- and y-axis;

xMin, xMax, yMin, and yMax define the real-world plot window; width and height define the screen dimensions of the plot in pixels; xDiv and yDiv give the number of divisions in the x and y directions where 0 values disable the grid; xSubDiv and ySubDiv give the number of subdivisions in the x and y directions; xDec and yDec give the number of decimal places used for x and y labels; and NumberOfColours gives the maximum colours desired on a plotsubject to some limitations discussed later. Scatter plots (Figure 5) are special in that they are produced by plotting character symbols at each graph point instead of joining the adjacent graph points with line segments. These plots are described in more detail later. Figure 6 illustrates the difference between the real-world coordinate system and the pixel coordinate system inherent in the Amiga's graphic utilities. In effect, the real-world coordinate system defines the domain in which the plotted functions live. For example, the plotted sinusoidal wave in Figure 1 has an x-range (xMin to xMax) of 0 to 7 and a y-range (yMin to yMax) of -1 to 1. This real-world space is mapped onto the Amiga screen within the pixel space defined by the width and height parameters passed to the InitPlot function by the xyToCoords mapping function in Listing 2.

If InitPlot returns a TRUE value, the plot has been successfully initialized and the plot can be drawn using the PlotFx procedure. This routine takes two arguments: the first is the plot handle which was defined by the InitPlot function; the second is a function which takes a real number (x) and returns some value F(x). Eight different functions are defined in Listing 1: DampSine, Cubic, RandomSine, AmpModulated, Response, StockPrices, ProfitValues, and ProfitValues2. As you can see from these examples, the processing performed by F(x) varies somewhat depending on the kind of plot to be drawn. Normal, scatter, log, and semi-log plots can accept any continuous function; the bar chart and linear plot require a discrete function which returns one value for each bar or point on the linear plot.

Congratulations, you have successfully produced your first plot! And it took only two calls to routines in the PlotLibrary! Of course,

this first plot can be enhanced in a number of ways which include labelling the plotted minima and maxima, displaying an information box, changing text, line, and screen colours, adding your own x-axis labels, and plotting multiple functions on a single axis. I'll describe each of these special effects next.

Special Effects

I remember being in school and having to determine the local minima and maxima of functions by taking the first derivative of the function. Well, the LabelMinMax routine does exactly that and then labels the resultant points right on the plot. LabelMinMax takes the same arguments which you also passed to the PlotFx procedure. You usually call LabelMinMax right after calling PlotFx but the calling order could be reversed if you want the plotted curve to overlay the minimum/maximum point labels. LabelMinMax works with any of the plot types defined in the PlotLibrary. For the scholarly reader, the local minima and maxima are determined by looking for sign reversals of the derivative of the plotted functions. In school you solved this same problem by determining where the derivative (or slope) was zero; however, with computers, tests against zero don't always work so the sign reversal approach gives more consistent results.

The InformationBox routine provides a powerful mechanism for displaying details about the plotted function. The preceding plots demonstrate the use of the information box. To use the information box, define an array of strings (LineType is exported from PlotLibrary for this purpose) which contains enough elements to hold all the lines of display text. For example, to display five lines you would declare:
Message : ARRAY [1..5] OF LineType;
and initialize this array with the information to be displayed in the information box. This information box can be positioned in the upper or lower plot area and can be centered, left-justified, or right-justified in the plot region. Both the information box's background and outline colours are specified in the call to InformationBox; text colour is set by the SetTextColour routine described below. The ScientificPlot routine in Listing 1 whose plotted output is shown in Figure 1 demonstrates how to use the InformationBox routine.

Four routines handle the colour manipulation of the plotted output. SetColourMap changes which of the 4096 available colours get mapped to a subset of colours available for the plot screen. An eight-colour screen would allow you to select eight of the 4096 colours to be displayed at one time. Each call to SetColourMap sets one screen colour, so eight calls are necessary to fully define all the colours for the example screen although unused colours don't need to be mapped. Defining names for the colour indices helps you to remember which colour is mapped where. The declarations in Listing 1 define a set of constants like Red = 5 so that a call to SetColourMap(Plot, Red, 11, 0, 0) can be used with a colour name instead of a number. The last three arguments define the amount (0 to 15) of red, green, and blue, respectively, for screen colour six (the first colour is zero). These eight colours are used to define the text colour (SetTextColour), the plotted curve colour (SetPlotColour), and the grid colour (SetGridColour). In each case, these procedures take, as an argument, the colour number (from 0 to 7 for an eight-colour screen). A maximum of 32 colours can be used with a plot screen width of 320 pixels; 16 colours for a 640 pixel plot screen width.

The x- and y-axis titles are positioned using the CenterLabelX and CenterLabelY procedures. Both these routines automatically center the text horizontally for CenterLabelX and vertically for CenterLabelY. You only need to specify the vertical position (in pixels) for the x-axis label and the horizontal position (also in pixels) for the y-axis label. As Figure 6 illustrates, the horizontal pixel numbers increase from left to right and the vertical pixel numbers increase from top to bottom. Some experimentation may be required to perfectly position the titles, although the positions shown in the BarChart routine in Listing 1 (395 for x label and 15 for y label) give a good starting point. Note that these labelling routines are not restricted to axis titles but could also be used for centering other information anywhere on the plot.

LabelX and LabelY are two more general labelling procedures which give you control over both the x and y position of plot text. The only difference between the two is that LabelX produces a horizontal line of text while LabelY produces a vertical text display. The Amiga operating system (V1.3) does not allow text to be rotated so the vertical text is composed of unrotated characters, each offset vertically from the others. Rotating characters for the y-axis is a non-trivial task I'll leave for another time.

The last text labelling procedure, SetLabelRoutine, lets you override the numerical labels which are automatically generated by the PlotLibrary. The example in Figure 3 uses this routine to produce month labels for a bar chart. This example passes the BarLabels procedure (Listing 1) to the SetLabelRoutine just before calling the PlotFx procedure. The passed procedure must have an interface identical to the one shown for the BarLabels procedure. The first argument identifies the plot division to be labelled and the second argument returns the corresponding label string. Currently, only the x-axis labels can be overridden, although it should be fairly simple for you to add y-axis custom labels by duplicating the source code shown for the x-axis custom labels.

Another procedure, SetScatterPlot, overrides the line-drawing modes of the standard plots and allows a character to be plotted instead for each point of the original graph. The default character is an 'O' but you can change to any other character using the SetScatterChar routine. The plotted scatter character colours are changed with the SetPlotColour procedure. Figure 5 shows the output of the ScatterPlot procedure from Listing 1.

The BarChart procedure in Listing 1 illustrates a method of plotting two data sets on the same coordinate axes. The first bar chart is created normally as was outlined above with a slight twist: the SetPlotOffset routines is used to offset the bars by four pixels to the left of where they would appear by default. The second bar chart uses a second data set which is offset four pixels to the right of the default position using the same SetPlotOffset procedure just before calling the PlotFx routine. This bar chart also demonstrates the power of passing a function (ProfitValues or ProfitValues2) which is called during the plotting operation since colours are set dynamically as the graph is plotted. In this case the plot line colour is changed with SetPlotColour to produce black and grey bars for a positive profit and red and pink bars for a negative profit. Figure 3 shows the resultant bar chart.

The SetPlotOffset procedure is used to give pixel offsets to reposition the bar charts described above. A negative pixel offset passed to the xoff argument shifts the plotted function to the left; a positive offset shifts the plot to the right. A negative pixel offset passed to the yoff argument shifts the plotted function up; a positive yoff offset shifts the plot down.

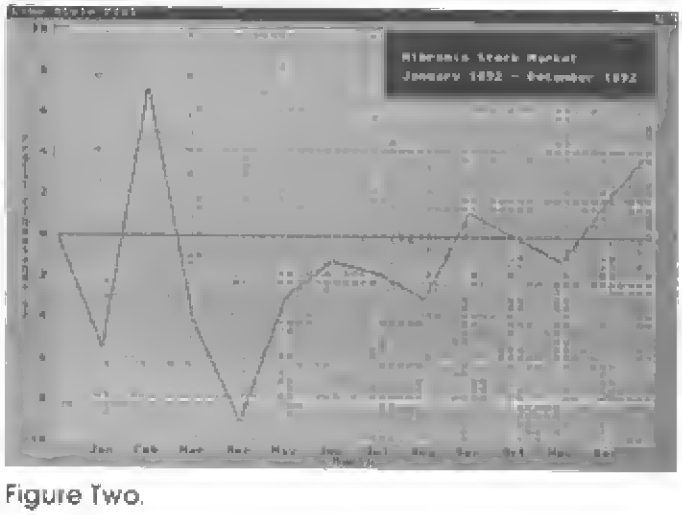
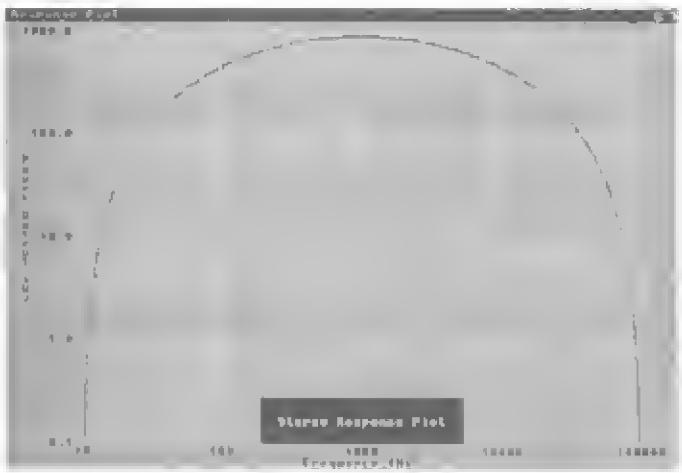
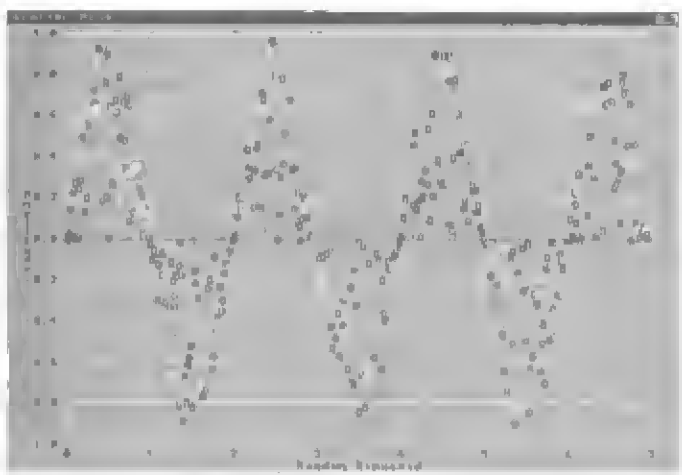


Figure Two.





From top to bottom, Figures Three through Five.

Any number of plots of any kind can be overlaid on the same plot as long as their real-world coordinate systems are equivalent (i.e., both bar charts had the same January to December x-axis and profit in thousands of dollars for the y-axis). Overlaid plots are usually not offset from each other except when you want to produce three-dimensional effects or bar charts. By adding offsets in the ShadowPlot routine in Listing 1, and changing the line colour, a plot with a shadow for emphasis is produced.

You now know how to overlay plots which share the same world coordinates, but what about producing plots which use different world coordinate systems? Fortunately, there is a simple solution if the SetPlotLimits procedure is used. This routine modifies the mapping of real-world coordinate points to the pixel coordinates required by the Amiga's display. By specifying different plot limits from those originally passed to the InitPlot routine, we can dynamically change our window size to accommodate larger or smaller plotted functions in the same plot window.

A related procedure called SetPlotScale in Listing 2 allows us to magnify the centre of the plot by an arbitrary amount.

The ScaledPlot procedure (Listing 1) provides an example which does just this. A cubic curve is shown at different magnifications all on the same set of axes (Figure 7).

The x- and y-axis labels correspond to the first plot and only represent a scaled version of the magnified plots. The plot scale factors passed to the SetPlotScale procedure must be values which are greater than 0.1. Numbers less than one actually reduce the plot size while numbers greater than one enlarge the plot. In this case the cubic function's x- and y-axis have been selectively magnified by two. The enlarged plots have been magnified more than can be shown in the plot window to demonstrate the line clipping algorithm used in the PlotLibrary.

As a final example, four independent plots (Figure 8) are shown on the same display screen. A procedure called InitOffsetPlot is used by the QuadPlot procedure in Listing 1 to place multiple scaled plots on the same screen. The initial steps are the same as before. A plot is initialized using InitPlot. This first step creates the canvas for the following four plots so the maximum number of colours and maximum pixel screen size need to be specified in this call. The second step requires a call to InitOffsetPlot with a reduced pixel area about 1/4 of the total screen area originally specified to give room for four plots. The call is to a new routine:

```
PROCEDURE  InitOffsetPlot(  VAR
Plot       : PlotType;
PlotKind   : PlotKindType;
xMin, xMax, yMin, yMax  : REAL;
width, height  : INTEGER;
xDiv, yDiv  : CARDINAL;
xSubDiv, ySubDiv  : CARDINAL;
xDec, yDec  : CARDINAL;
xOffset, yOffset  : CARDINAL;
OldPlot    :PlotType)
        : BOOLEAN;
```

The three last arguments in this procedure differentiate this routine from the call to InitPlot where xOffset and yOffset specify the plot offset relative to the pixel coordinate origin at (0,0) in the top left screen corner; and OldPlot is the plot variable initialized when InitPlot was called. The number of x and y divisions should also be cut in half since less text is visible in the reduced display area. The subdivisions have also been eliminated to avoid cluttering the smaller plots.

Enhancements

A number of enhancements can be made fairly easily to the PlotLibrary to extend its capabilities. These are:

1) Automatically determine the plot minimum/maximum points for the y-axis so the user doesn't have to know these values.
2) Allow text files containing columns to numbers to be used as the function to be plotted.
3) Use rotated and scaled text.
4) Implement a pie chart plot.

These new capabilities require a bit more work, but all these additions would extend the usefulness of this package so that it even rivals the commercial plotting systems. If there is enough interest, I'll tackle these additions in a future article.

Compiler Dependencies

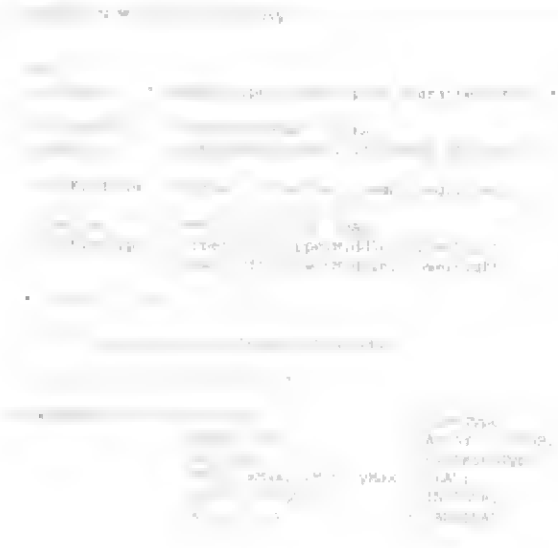
The PlotLibrary source code was compiled using the M2S compiler and any dependencies on this compiler have been flagged in the source. Specifically, a call to OpenRealTrans is required to grant access to the Amiga's transcendental floating point function library. Other compilers may provide this access automatically. All other Modula-2 specific calls should be available in other compilers. The Amiga system module names and functions may differ slightly in your compiler but should be similar to those used here. I have provided a StringUtils package to give a string length function and allow conversions from floating point to strings. Both these operations are included because they are required for the PlotLibrary and may not be provided with every Modula-2 compiler. If you have them available, feel free to substitute your own.

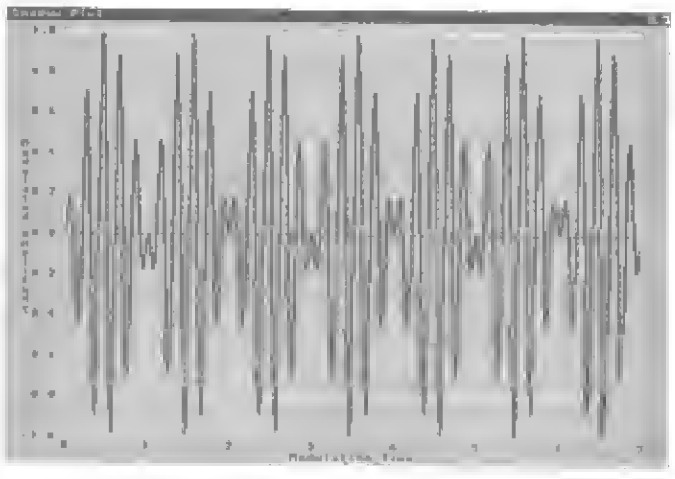
If your favourite language is C, you should have no problem translating from Modula-2 to C and several conversion programs can even perform the translation automatically.

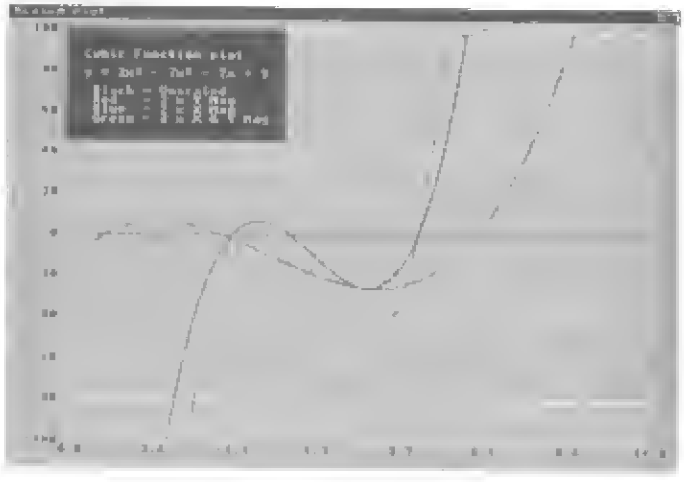
Conclusions

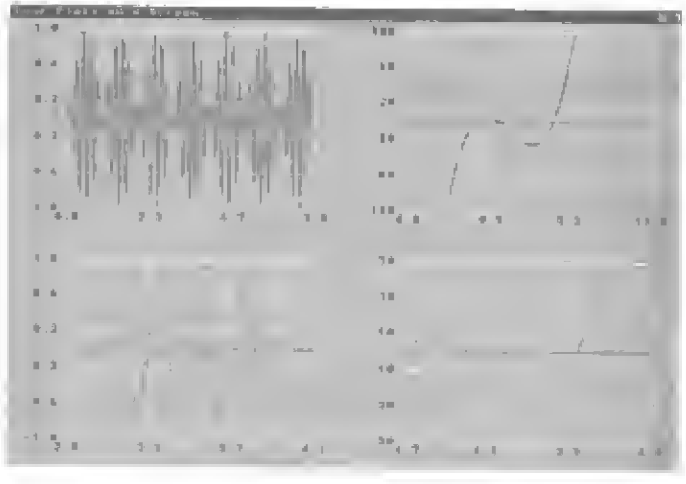
The PlotLibrary provides a set of routines which greatly simplifies the process of visualizing data for students, scholars, business executives, or experimenters. Simple plots can be produced with just two calls but more elaborate capabilities are available to produce an information box, label plot minima and maxima, place text and titles, overlay plots, and display multiple plots on a single screen. The built-in capabilities are powerful enough to unleash the imagination of the PlotLibrary user. Have fun!

Listing One









From top to bottom, Figures Six through Eight.

```
                    xAxDec, yDec                    : CARDINAL;
                    NumberOfCurves                  : CARDINAL;
                    BOOLEAN;

PROCEDURE SetOffsetPlot
                  (VAR Plot                         : PlotType;
                   PlotKind                         : PlotKindType;
                   xMin, xMax, yMin, yMax           : REAL;
                   width, height                    : INTEGER;
                   xDiv, yDiv                       : CARDINAL;
                   xSubDiv, ySubDiv                 : CARDINAL;
                   xDec, yDec                       : CARDINAL;
                   xOffset, yOffset                 : INTEGER;
                   OldPlot                          : PlotType)
                   : BOOLEAN;

PROCEDURE DonePlot(VAR Plot   PlotType) ;

PROCEDURE PlotFix(Plot : PlotType; Fx : PlotFunction);

PROCEDURE Empty     PlotType);

(*-------------------------------------------------------------------------
  Plot modification routines for more functionality
  -------------------------------------------------------------------------*)

PROCEDURE ClearPlot(Plot : PlotType);

PROCEDURE InformationBox(Plot          : PlotType;
                         Position      : PositionType;
                         Lines         : ARRAY OF LineType;
                         Fill,Colour   : CARDINAL;
                         OutlineColour : CARDINAL);


PROCEDURE PlotLine(Plot          : PlotType;
                   Colour        : CARDINAL;
                   x1, y1, x2, y2 : REAL);


PROCEDURE SetColourMap(Plot          : PlotType;
                       ColourIndex   : CARDINAL;
                       Red, Green, Blue : CARDINAL);


PROCEDURE SetPlotColour(Plot :     PlotType;
                        Colour : CARDINAL);


PROCEDURE SetPlotLimits(Plot                    PlotType;
                        xMin, xMax, yMin, yMax : REAL);


PROCEDURE SetPlotScale (Plot          : PlotType;
                        xscale, yscale : REAL);


PROCEDURE SetPlotOffset(Plot       : PlotType;
                        xoff, yoff : INTEGER);


PROCEDURE SetTextColour(Plot     PlotType;
                        Colour : CARDINAL);


PROCEDURE SetGridColour(Plot   : PlotType;
                        Colour : CARDINAL);


PROCEDURE SetZeroLine  (Plot   : PlotType;
                        On     : BOOLEAN );


PROCEDURE SetScatterPlot(Plot  : PlotType);


PROCEDURE SetScatterChar(Plot  : PlotType;
                         Ch    : CHAR);

(*-------------------------------------------------------------------------*)
```

```
                                                    ;

PROCEDURE (Plot : Label) Draw(x, y : INTEGER);

PROCEDURE CreateLabel(VAR Plot : PlotType;
                      xDec, yDec : CARDINAL;
                      NumberOfCurves : CARDINAL);

PROCEDURE SetLabelPlot(Plot : PlotType;
                       xLabel, yLabel : ARRAY OF CHAR);

PROCEDURE LabelX(Plot : PlotType;
                 Label : ARRAY OF CHAR;
                 x, y : INTEGER);

PROCEDURE LabelY(Plot : PlotType;
                 Label : ARRAY OF CHAR;
                 x, y : INTEGER);

PROCEDURE SetLabelMinMax(Plot : PlotType;
                         Fx : PlotFunction);

PROCEDURE SetLabelDirection(Plot : PlotType;
                            LabelDir : Direction; LabelPrec

END PlotLibrary.
```

Listing Two

(This listing is NOT complete as printed.)

```
IMPLEMENTATION MODULE PlotLibrary;

  (* Amiga-specific operating system modules *)
FROM Intuition   IMPORT ScreenPtr, NewScreen,
                        ScreenToBack, CloseScreen,
                        OpenScreen;
FROM Pens        IMPORT SetAPen, SetBPen, Move, Draw,
                        RectFill, AreaMove, AreaDraw;
FROM Rasters     IMPORT RastPortPtr, Text, TextLength;
FROM Text        IMPORT Text;
FROM Views       IMPORT ColorMap, ViewModeSet, ViewModes;

  (* General Modula-2 modules *)
FROM MathLib0    IMPORT ln, power;
FROM Storage     IMPORT ALLOCATE, DEALLOCATE;
FROM SYSTEM      IMPORT ADR, BYTE;

(* M2Amiga-specific module *)
FROM RealSupport IMPORT OpenRealTrans;

(* Simple routines available in RealSupport: PlotStr.Str, *)
FROM StringUtils IMPORT LengthStr, ConvRealToStr;


CONST
  BackColour  = 0;
  Colour1     = 1;
  Colour2     = 2;
  Colour3     = 3;

  CharHeight  = 8;
  CharWidth   = 8;


TYPE
  ClassType   = (Left, Right, Top, Bottom);
  ClassSet    = SET OF ClassType;
  PlotSet     = SET OF PlotKindType;

CONST
  XLog   = PlotSet{LogX, LogXY};
  YLog   = PlotSet{LogY, LogXY};


TYPE
  PlotType   = POINTER TO PlotRecord;
  PlotRecord
```

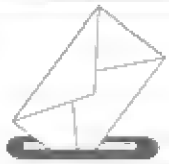
```
MODULE
  ( exec                        (*ExecPtr
  Graphics                      GetPortPtr
  Intuition                     SetFontPtr
  Workbench                     AbsExecBase
  InputDevice                   ARGINAL;
  Text                          ARGINAL;
  Layers                        ARGINAL;
  UserLabel;                    BOOLEAN;
  SetLabel;                     LabelProc;(* Label routine *
  ScatterPlot                   BOOLEAN;  (* Scatter plots *
  SetScatter                    ARRAY [0..
  DataLines                     BOOLEAN;  (* line on/off   *
  Minx, Miny, Maxx, Maxy        REAL;     (* real coords   *
  Xres, Yres, xMax, yMax        INTEGER;  (* screen outline*
  Xdim, Ydim                    INTEGER;  (* & dimensions, *
  Dxx, Dyy                                (* major divs.   *
  Dxxx, Dyyy                              (* minor divs.   *
  Xmax, Ymax                    ARGINAL;  (* decimal places*
  xscale, yscale                REAL;     (* x'y scaling   *
  xoff, yoff                    INTEGER;  (* plot offsets  *
  gridOnOff                     BOOLEAN;  (* draw grids    *
  Copy: TAPlot                  BOOLEAN;  (* not original  *
  ) ;

(*                                                          *)
(* The following are a number of utility routines used      *)
(* only within this module to facilitate the graphing.      *)

PROCEDURE CreateScreen(Width, Height, Depth : INTEGER;
                       HighResolution       : BOOLEAN;
                       Interlaced           : BOOLEAN;
                       Title                : ARRAY OF CHAR
                     ) : Screenptr;
VAR
  Screen    : NewScreen
  ViewMode  : ViewModeSet;
BEGIN
  ViewMode := ViewModeSet{};
  IF HighResolution THEN INCL(ViewMode, HiRes) END;
  IF Interlaced     THEN INCL(ViewMode, Lace) END;
  WITH Screen^ DO
    LeftEdge      := 0;
    TopEdge       := 0;
    Width         := Width;
    Height        := Height;
    Depth         := Depth;
    DetailPen     := BYTE(1);
    BlockPen      := BYTE(3);
    ViewMode      := ViewMode;
    Type          := CustomScreen;
    Font          := NIL;
    DefaultTitle  := NIL;
    Gadgets       := NIL;
    CustomBitMap  := NIL;
    IF Title[0] # 0C THEN
      DefaultTitle := ADR(Title);
    END;
  END;

  RETURN OpenScreen(ADR(Screen^));
END CreateScreen;


PROCEDURE Round(x : REAL) : INTEGER;
BEGIN
  RETURN TRUNC(x + 0.5);
END Round;


PROCEDURE Power(x, y : REAL) : REAL;
BEGIN
  RETURN power(10.0, x);
END Power;


PROCEDURE Log0(x : REAL) : REAL;
BEGIN
  IF x <= 0.0 THEN (* capture illegal values *
    RETURN 0.0;     (* and return log(1)     *
  END;
  RETURN ln(x);
END Log0;


PROCEDURE xyToCoords(Plot       : PlotType;
                     x, y       : REAL;
                     VAR xG, yG : INTEGER);
(* Convert floating point world coordinates to graph space
   coordinates. *)
BEGIN
  WITH Plot^ DO
    xG := Round((x-Minx)*FLOAT(Maxx-Xorg)/(Maxx-Minx)+Xorg);
    yG := (YMax-Round((y-Miny)*FLOAT(YMax-Ymin)/(Maxy-Miny)));
  END;
END xyToCoords;


PROCEDURE CoordsToxy(Plot       : PlotType;
                     xG, yG     : INTEGER;
                     VAR x, y   : REAL);
(* Convert graph space coordinates to floating point world
   coordinates. *)
BEGIN
  WITH Plot^ DO
    x := FLOAT(xG-XMin)*(Maxx-Minx)/FLOAT(XMax-XMin)+Minx;
    y := FLOAT(YMax-yG)*(Maxy-Miny)/FLOAT(YMax-YMin)+Miny;
  END;
END CoordsToxy;


PROCEDURE SetScreenOffset(Plot : PlotType; dx, dy : INTEGER);
(* Offset the graph point by the plot amounts *)
BEGIN
  INC(x, Plot^.xoff);
  INC(y, Plot^.yoff);
END OffsetPoint;


PROCEDURE ClipLine(Plot                 : PlotType;
                   Color                : ARGINAL;
                   x1, y1, x2, y2       : INTEGER);
(* Line clipping algorithm - Two points are passed (x1, y1)
   and (x2, y2) to be drawn in the color specified. The
   line are passed (x1, y1) and (x2, y2) is to be
   clipped within the Plot area. NOTE: The Amiga screen
   coordinates have y = 0 at the top of the screen.

   From an algorithm published in "Microcomputer Graphics"
   for the IBM PC" by Roy E. Myers. *
VAR
  x, y      : ARRAY [1..2] OF INTEGER;
  xi, xm    : INTEGER;
  t1, t2    : INTEGER;
  offset    : INTEGER;
  Bound     : ARRAY [1..2] OF ClassSet;
  Discard   : BOOLEAN;
  i         : CARDINAL;

  PROCEDURE Classify (VAR boundary : ClassSet;
                          x, y     : INTEGER);
  BEGIN
    boundary := ClassSet{}; (* assume inside by default *
    WITH Plot^ DO
      IF x < xn THEN
        INCL(Boundary, left);   (* x is outside left boundary *
      ELSIF x > xm THEN
        INCL(Boundary, Right);  (* x is outside right boundary *
      END;
```

—The complete Source Code and executable files may be found on the AC's TECH Disk

Please write to:
Michael Griebling
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

List of Advertisers

MOVING?



SUBSCRIPTION PROBLEMS?

Please don't forget to let us know. If you are having a problem with your subscription or if you are planning to move, please write to:

Amazing Computing Subscription Questions
PiM Publications, Inc.
P.O. Box 2140
Fall River, MA 02722

Please remember, we cannot mail your magazine if we do not know where you are.

Please allow four to six weeks for processing

AC's TECH Disk
Volume 2, Number 4

A few notes before you dive into the disk!

- You need a working knowledge of the AmigaDOS CLI as most of the files on the AC's TECH disk are only accessible from the CLI.

- In order to fit as much information as possible on the AC's TECH Disk, we archived many of the files, using the freely redistributable archive utility 'lharc' (which is provided in the C: directory). lharc archive files have the filename extension .lzh.

  To unarchive a file *foo.lzh*, type *lharc x foo*
  For help with lharc, type *lharc ?*
  Also, files with "lock" icons can be unarchived from the WorkBench by double-clicking the icon, and supplying a path.

Be Sure to Make a Backup!

CAUTION!

Due to the technical and experimental nature of some of the programs on the AC's TECH Disk, we advise the reader to use caution, especially when using experimental programs that initiate low-level disk access. The entire liability of the quality and performance of the software on the AC's TECH Disk is assumed by the purchaser. PiM Publications, Inc., their distributors, or their retailers, will not be liable for any direct, indirect, or consequential damages resulting from the use or misuse of the software on the AC's TECH Disk. (This agreement may not apply in all geographical areas.)

Although many of the individual files and directories on the AC's TECH Disk are freely redistributable, the AC's TECH Disk itself and the collection of individual files and directories on the AC's TECH Disk are copyright © 1990, 1991 by PiM Publications, Inc. and may not be duplicated in any way. The purchaser is encouraged to make an archive/backup copy of the AC's TECH Disk.

Also, be extremely careful when working with hardware projects. Check your work, twice, to avoid any damage that can happen. Also, be aware that using these projects may void the warranties of your computer equipment. PiM Publications, or any of it's agents, is not responsible for any damages incurred while attempting this project.

Programming the Amiga in Assembly Language

by William P. Nee

USING LHARC

Always on the lookout for an easier way to do things, I want to show you a new way to load the files you need for assembly language programming into RAM. I put LHARC, located on the magazine disk, in the C directory and then used it to create two files containing:

```
C/CFILES.LZH          INCLUDES.LZH
A68K    BLINK         EXECMACROS.i
CD      COPY          DOSMACROS.i
DELETE  DIR           INTMACROS.i
ED      MAKEDIR       GFXMACROS.i
MORE                  ASB   DCOPY   DSAVE
```

Then I modified my Startup-Sequence to include:

```
COPY CFILES.LZH RAM:C
COPY INCLUDES.LZH RAM:
COPY C/LHARC RAM:C
CD RAM:C
LHARC X CFILES.LZH
DELETE CFILES.LZH
CD RAM:
RAM:C/LHARC X INCLUDES.LZH
DELETE INCLUDES.LZH
```

This method does use a lot of memory since you have both the packed and unpacked files in RAM for a while. If you're running low on memory, include less in the LZH files, or try deleting each packed file individually after you unpack it, instead of deleting the entire file when all of the individual files are unpacked. I've also included NoClick3.4, an excellent program that stops the drives from clicking all the time. You might want to include it in your Startup-Sequence.

WALLPAPER

In this article I'll discuss double-buffering (using multiple screens to swap images), running assembly language programs from Basic, and passing values to a program. I'll start with double-buffering first and use a program called WALLPAPER to demonstrate it. This program sets cells in a 160X160 array to various values. During the second pass the values of the four neighbors around a cell are added together and ANDed with 3; this becomes the new cell value for array2. When new cell values have been computed for the entire array2, they're transferred to array1 and shown on the screen.

Without any further modification, running the program this way would show a noticeable ripple effect. But, with multiple screens, we can show one screen, compute and place points on the second screen, and display it; then compute and place new points on the first screen and display it. Keep alternating between screens until the LMB (Left Mouse Button) is pressed.

To set up multiple screens we need to open two screens and two windows. We'll also need the bitplane addresses for each window since I'll be using the QPSET macro from the previous article of this series. The key to double-buffering is the Intuition command ScreenToFront. Using this command we can show one screen (bring it to the front) and use a different RastPort to draw on a different screen; then bring that screen to the front and use the other RastPort to draw. Keep looping in this way. A generic example using SCREEN1, RP1, SCREEN2, and RP2 is:

```
LOOP
  USE RP1
  SCREEN2 TO FRONT
  PCLS
  DRAW
  USE RP2
  SCREEN1 TO FRONT
  PCLS
  DRAW
BRANCH TO LOOP
```

Now let's look at Listing 1. The two INCLUDE files will handle the Exec and Graphics macros. Since the Intuition macros are so different because of double-buffering, it's easier to include them as part of the listing. The program first opens SCREEN1 and SCREEN2, and then opens two windows. The main difference between the two windows is that WINDOW1 is set to be active (ACTIVATE) when it is opened.

After each window is opened, its RastPort is obtained, saved, and then used to get five bitplane locations. The bitplanes are only accurate according to the DEPTH (1-5) you set at the beginning of the program. Next, the ARRAY macro will clear two arrays to zero; their size is again defined at the start of the program as SIZE. There are two Demos for you to use; the first draws a cross and the second a box.

The next part of the program begins the double-buffering routine. SCREEN1 is brought to the front (it's blank) while RastPort2 is used as the current RastPort. After clearing

PART IV - A Return To Basic

SCREEN2, new array values are computed and drawn on SCREEN2 using the QPSET macro. Then SCREEN2 is brought to the front displaying it, and RastPort1 is assigned as the current RastPort. SCREEN1 is cleared, new array values are computed, and then drawn on it. A check is made to see if you pressed the LMB and, if not, the entire process repeats.

When you do press the LMB, the memory in both arrays is freed, both windows and then both screens are closed, and the Graphics and Intuition libraries are closed. In the data section you can see that I reserved space for the screens, windows, RP1, RP2, and an RP. By moving the current RP1 or RP2 to RP, I was able to use all the macros that call for RP. Save this program as WALLPAPER.ASM and assemble and BLINK it as WALLPAPER—or copy it from the magazine disk. If you don't have them from Part III of this series, the two INCLUDE macro files are also on the magazine disk. I've made a few changes in the GFXMACROS.i. Try changing this program by increasing

have the machine language program store the color value of all set points in an array just like Basic's POINT command. In fact, I call the macro using ReadPixel POINT.

The arrays are 160X160 so I used 26000 as the SIZE in the ARRAY macro. Another new macro is PUSHALL which saves all eight data registers along with the first seven address registers and stores their values just below the Stack Pointer (a7). MOVEM means to move multiple registers; the "-" means through and "/" means also. So we'll move multiple registers d0 through d7 and also a0 through a6 to the memory location starting four bytes below the Stack Pointer. The macro PULLALL reverses the procedure, returning the original value of all the registers. The only other new macro, as I mentioned, is POINT which uses the ReadPixel routine to return the color value of every point on the screen within the array. Since Basic considers the border when setting a point and FASTPIXEL doesn't, any point you draw in Basic is four off in the across

To set up multiple screens we need to open two screens and two windows. We'll also need the bitplane addresses for each window since we'll be using the QPSET macro from the previous article of this series.

the DEPTH value and then ANDing the neighborhood SUM with a different value.

FROM BASIC

Next, I'll discuss passing values to and running your machine language program from Basic. The general idea is to convert the assembled program into a new file of data lines that are the actual number values (0-255) of each instruction. This file is then merged with your Basic program, and each data line is READ and stored in a variable array. The machine language program is executed with a CALL to the beginning location of the array. I'll go over this with you step-by-step as we develop the program.

Let's look at the assembly language portion, Listing 2 first. The program is a lot smaller since we're using Basic to set up the screen and window. There is one new graphics routine, ReadPixel, which will read the color value of any point on the screen and return that value in register d0. This will allow you to draw a picture on the screen or set any points, and then

direction and two off in the down direction. That's why POINT only adds 76 to across instead of the usual 80 to center the display, and only adds 18 to down instead of 20.

The ARRAY macro uses CLEAR and PUBLIC memory since we're not sure where the memory will be found now that we're running everything from Basic. PUBLIC lets the computer choose FAST memory if available or CHIP memory as an alternative. Since the memory is used only to store data, we don't care where it gets assigned. This array uses the rules for the game of *Life*—a computer game where there are no winners or losers. At the start, a cell in the array is either alive (set to 1) or dead (set to 0). During the first generation the neighborhood value of the eight cells around a center cell is computed. If the original cell is alive and has only two or three neighbors, it will live to the next generation; if it is dead but has three neighbors, it will become alive for the next generation. Any other case results in a dead cell. We'll examine these rules more closely later.

Now let's review the entire machine language program. In previous articles we got the bitplane addresses by using the RastPort address within the window structure. But now we've let Basic open the window, so where is the window structure or the RastPort? In Basic, the window structure is returned with WINDOW(7) and since it is used so often, the RastPort as WINDOW(8). This location will be passed from Basic when we call the machine language program. When only one variable is passed, its value is in register d7 at the start of the program. Before you did anything else you would store this value in RP and get the bitplanes in the usual manner.

PASSING VARIABLES

I said that I'd show you how to pass multiple variables from Basic to your machine language program. Using the PUSHALL macro I described, we'll move 15 four-byte registers below the Stack Pointer altogether filling bytes 0-63. Any variables you pass with your call start at the next location (byte64) and will be four bytes long. So if you pass variables A, B, and C within your call, variable A=64(a7), B=68(a7), and C=72(a7). These locations can be used whenever you want within a program since they won't change. The assembler I'm using, A68K, usually requires you to move a variable to a labeled location before you use it. If your call passed three variables named RP, ACROSS, and DOWN, you would probably save them as:

```
MOVE.L  64(A7),RP
MOVE.L  72(A7),DOWN
MOVE.L  68(A7),ACROSS
```

The first variable, as I said, is also initially stored in register d7.

Next, the Graphics library is opened. Since Basic may store our assembled program anywhere, the location called "gfx" may always be different. But we can't re-assemble the program each time we run it, so we must use what is called "position independent code." The Position Counter (PC) is another register that keeps track of the address of each line being executed so a line like GFX(PC) is just an offset. When this line is assembled it will resemble a branch and will always store the address a certain number of bytes away from the current address in a1 instead of storing the original address of "gfx". This will generate accurate code no matter where Basic stores the library names.

Next, the program reads each point on the screen within the array and stores that value in array1; this takes a few seconds to do. To compute the neighborhood of a cell, I said that we added up the values of all eight cells around it. Most of the cells in the early generation of life have a zero value, so adding up eight zeros just to get zero wastes a lot of time. Another method that works much more quickly is to add the current value of a cell to the eight corresponding neighbor cells in the second array. That way you only stop for cells that have a value and skip over any dead cells. At the end of the first pass, array2 will contain the value of each neighborhood for the corresponding cell in array1. This routine would look like:

```
TST.B   (A4)
BEQ.S   CONTINUE
ADDQ.B  #1,-162(A5)  ;top-left neighbor
ADDQ.B  #1,-161(A5)  ;top neighbor
ADDQ.B  #1,-160(A5)  ;top-right neighbor
ADDQ.B  #1,-1(A5)    ;left neighbor
etc.
```

RULES OF LIFE

Let's look at the rules again for the next generation, restating them and using an OR table to show the results.

CURRENT	NEIGHBORHOOD	NEW	OR TABLE
0	3	1	0 OR 3 = 3
1	2	1	1 OR 2 = 3
1	3	1	1 OR 3 = 3

The only cases that produce a new cell are when the old cell's value OR'd with the neighborhood value results in 3. Any other result will create a dead cell. This will give us just one comparison instead of three.

I also wanted my version of Life to wrap-around. In this way, a shape going off the right side will reappear on the left; a shape going off the bottom will reappear at the top. We'll still use our 160X160 array. The inside square starting at (1,1), the center square, uses the same routine as Listing 1. Any location within an array is (length+1)*down+across. That's why (1,1) or (160+1)*1+1 is the 162nd element within the array.

WRAP-AROUND

There are two ways to approach programming a wrap-around function. If we limit the array size to multiples of 2-1 (i.e., 31, 63, 255), any cell location AND the length will always wrap back. In a 63X63 array, the cell to the right of an edge would be (63+1) AND 63 or the 0 cell in that row; the cell above a top cell would be (0-1) AND 63 or the 63rd cell in that column. But this does restrict the size of the array and, as a minimum, the cells on the outside rows and columns must always have the extra AND which slows down the program a little.

I decided to actually compute and check the eight squares around the four corners, the top row, the bottom row, and the sides. Rather than repeatedly typing "addq.b #1" , we'll let the INCREASE macro do that and just pass the location of the eight cells within array2 that get increased. Starting with the cell in the upper-left corner, its first neighbor is one cell to the left, which wraps to the right of the first row, and one up, which wraps to the bottom of the last column—the lower-right corner. That location is 161*160+160 or 160*162 and that's the first value passed in the "increase" macro. The next neighbor is one above which wraps to the bottom of the first column or 160*161+0. All eight neighbors are computed in this manner.

The next corner is 160 away, and if it's alive, compute its eight neighbors and increase each of them in array2. The lower-left corner is 161*160 cells away, so check it and, if necessary, its eight neighbors. Finally, the lower-right corner is 162*160 cells away. Notice that we never had to change the location in registers A4 and A5. They remained constant while we kept looking at and increasing locations a fixed distance from them.

To get the neighbors for the top row, however, it will be easier to increase our locations within the arrays by 1 after each

check. The top row starts at (1,0) and is 159 bytes long (1-159). The DBF counter is set to 158 since it goes down to 0. Whenever possible, use the postincrement mode to increase a register rather than LEA. After the neighbors for the top row are computed, registers A4 and A5 are changed to (1,161) for the bottom row.

I combined the left and right side since an increase of 1 from the right side takes us back to the left side. After all the cells on the sides have been checked and their neighbors in array2 increased, the program looks at all the cells within the center square. This is our familiar procedure from previous listings where we start at location (1,1) in both arrays.

When the entire array has been checked, array2 contains the sum of the neighbors around each corresponding cell in array1. Store that value in "sum" and then clear array2. Now we need to OR this value with the original cell, so store the contents of array1 in d1 then OR d1 with "sum". If the result is not 3 the new cell is dead, so store a 0 in "sum". But if the value is 3 the new cell is alive regardless of its previous state, so store a 1 in "sum". Compare the new value of "sum" to the old value in array1. If they're the same we can go to the next location. If they're different, however, we must change the value in array1 and reset the point on the screen.

Since SUM is the current value, store that in array1 for the new cell value. The FASTPIXEL routine is slightly different from previous ones since there is only one bitplane and there is no need for a loop through the planes. Once the location that must be changed is computed, we can change it with just one command. Because it can only be 0 or 1 and must be changed, use the BCHG (Bit CHanGe) command to change, or reverse, one bit. Then the array1 address is increased by 1, all the rest of the values in array2 compared to array1, and the necessary bits changed. If you don't press the LMB, the program branches back to the beginning.

DATA LINES

Save this program as LIFE.ASM then assemble and BLINK as LIFE, or copy it from the magazine disk. Now we have to transfer the assembled program into a file of numbers that can be made into DATA lines. Listing 3 is a Basic program that does this. Save Listing 3 as DATA and run it from Basic, or even better, transfer this program and AmigaBasic to RAM and run it from there. At the prompt, enter the program name LIFE. A new ASCII file called LIFE.DAT will be created consisting of data lines with eight numbers per line; the last number "999" is added to indicate the end of data.

Now type in Listing 4 and then MERGE "LIFE.DAT"; save this program as LIFE.BAS,a—or copy it from the magazine disk. The program uses P() to hold the data value and assigns values to PROGRAM and VALUE. By doing this at the beginning, Basic will assign them an immediate location and not move them around later. RP& is the RastPort and PROGRAM is the beginning of the array holding the machine language data. There are several demos to use; the R-pentano will increase for 1000 generations before it stabilizes. Notice that it produces little objects called "gliders" that slowly move across the screen. The gliders will always live unless they bump into something. The collision of two gliders can produce some interesting displays.

As I mentioned, A68K adds some extra code at the beginning and end of its assembled program. To see where the program starts, look for the code of the PUSHALL macro (72, 231, 255, 245). This is usually around line 4 of the DATA lines. Wherever it is, though, all code before it can be deleted. If you put in a few NOP's at the end of the program, then the code of 78,113 will stand out and you can eliminate everything after them except for the "999". This won't make the program run any differently, but it does cut down on the number of DATA lines and the size of the array you need to hold them. Now, whenever you use the Basic LIFE program, the machine language code is already there for you. Notice that there is no CALL command, just the name of the array and the value to be passed. This is an alternative way of using the CALL command.

NEXT TIME

In the next article I'll discuss menus and gadgets in machine language and we'll use double-precision floating-point numbers to draw the Mandelbrot Set. If you have any questions about these articles or ideas for future ones, you may write to me in care of this magazine.

Listing One

```
;LISTING 1
   nolist
   include  execmacros.i
   include  gfxmacros.i
   list
ww.screen      $2e
ww.rport       $32
nw.screen      $1e
borderless     $800
activate       $1000
depth    equ   2
sum      equ   d7
```

```
across   equr  d6
down     equr  d5
rp.oper  = 51b
size     = 26000

openscreen  =  -198
openwindow  =  -204
closescreen =  -66
closewindow =  -72
screentofront  =  -252

intlib  macro
   movea.l  intbase,a6
   jsr   \1(a6)
   endm
openscreen  macro
   lea   \1,a0
   intlib   openscreen
   move.l   d0,\2
   endm
openwindow  macro
   lea   \1,a0
   move.l   \2,nw.screen(a0,
   intlib   openwindow
   move.l   d0,\3
   endm
start:
   move.l sp,stack                  ;save stack pointer

   openlib  gfx,done2          ;open all the libraries
   openlib  int,done2

set_up
   openscreen  myscreen1,screen1
   openscreen  myscreen2,screen2

   openwindow  mywindow1,screen1,window1
   move.l   d0,a0
   movea.l  ww.rport(a0),a1
   move.l   a1,rp2
   movea.l  4(a1),a1
   lea      bp1,a0
   move.l   8(a1),(a0)+
   move.l   12(a1),(a0)+
   move.l   16(a1),(a0)+
   move.l   20(a1),(a0)+
   move.l   24(a1),(a0)

   openwindow  mywindow2,screen2,window2
   movea.l  d0,a0
   movea.l  ww.rport(a0),a1
   move.l   a1,rp2
   movea.l  4(a1),a1
   lea      bp2,a0
   move.l   8(a1),(a0)+
   move.l   12(a1),(a0)+
   move.l   16(a1),(a0)+
   move.l   20(a1),(a0)+
   move.l   24(a1),(a0)

;   mode  jam1
memory
   array    array1,done1
   array    array2,done1

;   bra.s    demo2

demo1:
   moveq    #0,across
   movea.l  array1,a4
   lea      12910(a4),a4
   movea.l  array1,a5
   lea      4910(a5),a5
d1a move.b   #1,0(a4)
   lea      1(a4),a4
   move.b   #1,0(a5)
   lea      761(a5),a5
   addq.w   #1,across
   cmp.w    #101,across
   bne.s    d1a
   bra.s    loop1

demo2:
   moveq    #0,across
   movea.l  array1,a3
   lea      8100(a3),a3
   movea.l  array1,a4
   lea      8160(a4),a4
   movea.l  array1,a5
   lea      8100(a5),a5
   movea.l  array1,a6
   lea      17760(a6),a6
   moveq    #0,across
d2a move.b  #1,0(a3)
   lea      1(a3),a3
   move.b   #1,0(a4)
   lea      161(a4),a4
   move.b   #1,0(a5)
   lea      161(a5),a5
   move.b   #1,0(a6)
   lea      1(a6),a6
   addq.w   #1,across
   cmp.w    #61,across
   bne.s    d2a

   moveq    #0,across
   moveq    #0,down
;double buffering
loop1
   movea.l  screen1,a0
   intlib   screentofront
   movea.l  rp2,a0
   move.l   a0,rp
   pcls
   lea.l    bp2,a0
   bsr      routine

   movea.l  screen2,a0
   intlib   screentofront
   movea.l  rp1,a0
   move.l   a0,rp
   pcls
   lea.l    bp1,a0
   bsr      routine

endloop:
   inb      loop1
   bra      done1
routine
   movea.l  array1,a4
   lea      162(a4),a4
   movea.l  array2,a5
   lea      162(a5),a5
   moveq    #0,down
   moveq    #0,across
   move.w   #158,down
```

```
r2 move.w   #158,arr
r1 moveq    #0,sum
   move.b   )61(a4),sum
   add.b    -1(a4),sum
   add.b    +1(a4),sum
   add.b    161(a4),sum
   andi.b   #3,sum
   move.b   sum,(a5)+
   lea      1(a4),a4
   dbf      across,r1
   lea      2(a4),a4
   lea      2(a5),a5
   dbf      down,r2

   movea.l  array1,a4
   movea.l  array2,a5
   moveq    #0,down
r4 moveq    #0,across
r3 cmp.b    (a5)+,(a4)+
   bne      nextone
   move.b   -(a5),sum
   move.b   sum,-(a4)
   qpset    sum,80,20
   lea      1(a4),a4
   lea      1(a5),a5
nextone
   addq.w   #1,across
   cmp.w    #161,across
   bne.s    r3
   addq.w   #1,down
   cmp.w    #161,down
   bne.s    r4
   rts
done1:
   free     array2
   free     array1

   movea.l  window2,a0
   intlib   closewindow
   movea.l  window1,a0
   intlib   closewindow
   movea.l  screen1,a0
   intlib   closescreen
   movea.l  screen2,a0
   intlib   closescreen
done2:
   closelib    gfx
   closelib    int
   move.l      stack,sp
 rts
   evenpc
stack    dc.l  0
gfxbase  dc.l  0
intbase  dc.l  0
screen1  dc.l  0
screen2  dc.l  0
window1  dc.l  0
window2  dc.l  0
rp       dc.l  0
rp1      dc.l  0
rp2      dc.l  0
array1   dc.l  0
array2   dc.l  0
bp1   ds.l  6
bp2   ds.l  6
gfx      dc.b  'graphics.library'
   evenpc
int      dc.b  'intuition.library',0
   evenpc
myscreen1
 dc.w 0,0,320,200,depth
 dc.b 0,1
 dc.w 0
 dc.w $f
 dc.l 0,0,0,0
   evenpc
myscreen2
   dc.w  0,0,320,200,depth
   dc.b  0,1
   dc.w  0,$f
   dc.l  0,0,0,0
   evenpc
mywindow1
 dc.w 0,0,320,200
 dc.b 0,1
 dc.l 0
 dc.l borderless!activate
 dc.l 0,0
 dc.l 0
 dc.l 0,0
 dc.w 0,0,0,0
 dc.w $f
   evenpc
mywindow2
   dc.w  0,0,320,200
   dc.b  0,1
   dc.l  0
   dc.l  borderless
   dc.l  0,0
   dc.l  0
   dc.l  0,0
   dc.w  0,0,0,0,$f
   end
```

All the source code and executable files for Programming the Amiga in Assembly Language can be found on the AC's TECH Disk.

Please write to:
William Nee
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

The Joy of Sets

by Jim Olinger

TWO EVENTS SPURRED MY INTEREST in Mandelbrot sets. For several years I had been vaguely aware of a recently-discovered branch of mathematics known as "fractals", which require so many computations that they can only be calculated with a computer, and I knew a Mandelbrot set was a specific kind of fractal.

However, smashing into engineering math in college (and eventually emerging with a computer science, not engineering, degree) left me with a distaste for high-level mathematics. I was also turned off by the computation-intensive nature of the field. Between 1989 and 1991, *Amazing Computing* published a six-part series on fractals by Paul Castonguay. The articles were interesting, but it appeared that a typical fractal would require four or five hours to generate. With the Amiga's multitasking, the calculations could be run in the background, but it would slow the system down and my software development efforts typically resulted in several system crashes in that amount of time.

Then I bought a 25 MHz. Amiga 3000 and read The Ghost From The Grand Banks by Arthur C. Clarke. The novel describes two corporations in 2012 racing to raise the Titanic for the 100th anniversary of her sinking. A major subplot is the Mandelbrot set obsession shared by several of the characters.

"Mandelmania", as Clarke calls it, takes several forms in *Ghost*, from a grief-stricken mother spending months searching a supercomputer-generated section of a set with a base image larger than the galaxy for a lost child, to the construction, with tragic results, of a Mandelbrot set-shaped

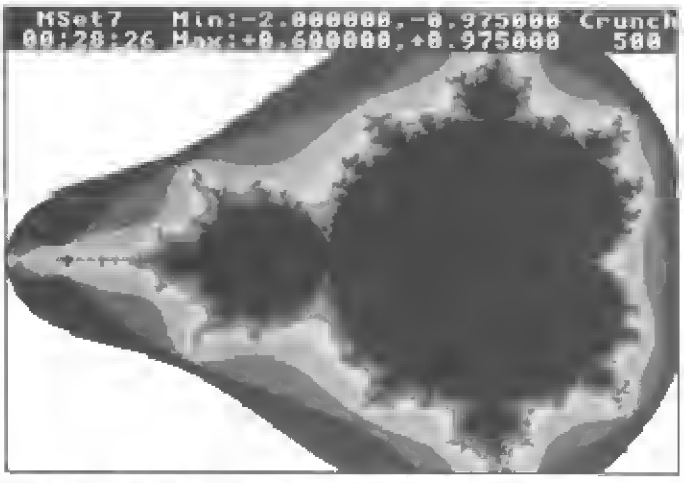


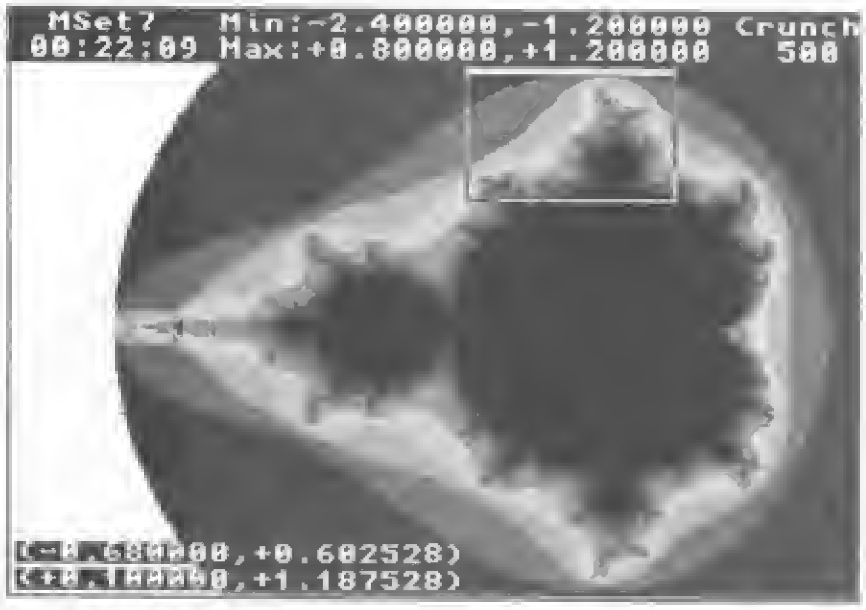
Figure One: The base Mandelbrot set.
Page 57: Top, Figure 2A. Bottom, Figure 2B.
Each image is a "zoom" of the previous image.

pond on the grounds of an Irish castle. Clarke himself doesn't appear to be immune. Some of his descriptions of Mandelbrot sets have a poetic quality I've never seen in his other work:

"The image expanded. It was like watching a zip-fastener being pulled open - except that the teeth of the zipper had the most extraordinary shapes. First they looked like baby elephants, waving tiny trunks. Then the trunks became tentacles. Then the tentacles sprouted eyes, which opened up into black whirlpools of infinite depth."

"They swept past the whirlpools, skirting mysterious islands guarded by reefs of coral. Flotillas of seahorses sailed by in stately procession. At the screen's exact center, a tiny black dot appeared, expanded - and revealed itself as an exact replica of the original set. Or was it?"

Suddenly, I wanted to get in there and do some exploring of my own. My shiny new A3000 isn't quite a Connection Machine, but it has a *lot* more computing horsepower then Benoit Mandelbrot had when he started his work on fractals in the 1970s. So, I dug through my back issues of *Amazing Computing* for Paul Castonguay's articles and started programming.



Thirty Seconds Over Lake Mandelbrot

A detailed explanation of the mathematical theory behind Mandelbrot sets is beyond the scope of this author. For more technical information, I strongly recommend Paul Castonguay's Amazing Computing articles and Chapters 15, 20 and the appendixes of The Ghost From The Grand Banks, where Clarke mentions the Amiga 2000 he used in his Mandelbrot set research (He now has an Amiga 3000). In case you don't have these resources available, I will briefly discuss a few basic Mandelbrot set concepts.

The Mandelbrot set is based on an iterative process. The Mandelbrot equation is solved for an initial value. Then, the output of the equation is fed back into its input and the equation is recalculated until its output either rapidly increases toward infinity ("blows up") or starts producing a single value or a small range of values ("converges").

The Mandelbrot "equation" is actually two equations: "Xnew = x * x - y * y + i" and "Ynew = 2 * x * y + j". These are extremely simple, using only addition, subtraction and multiplication. Most mathematically knowledgeable people who haven't been exposed to fractals would guess that the equations would produce some sort of oval. Anything truly complex should require higher mathematics, such as logarithms or trigonometric functions. No one, not even Benoit Mandelbrot when he started his research, could imagine the complexity hidden within those simple equations.

A mathematical set is a list of solutions to an equation. The Mandelbrot set is the boundary between the numbers on an X-Y plane that blow up and those that converge.

The limits of the Mandelbrot set are -2.0 to +0.8 in the X direction and -1.2 to +1.2 in the Y direction. This is called the "base set". Any values outside this range will quickly blow up. The lower X limit of the default "base set" in the program presented in this article is -2.4. This reveals the complete lack of features to the left of -2.0 and shows the detailed structures at the top and bottom of the screen.

The values used in Mandelbrot set calculations are pure numbers. They do not represent any kind of physical quantities. However, physical terms are used for convenience. Compass directions describe relative positions within the set. For example, an X value of -2.0 is "Utter West". This is the edge of the Mandelbrot universe and nothing exists

beyond this point. The large areas of convergence are often called "oceans" or "lakes" and the smaller areas around them are "bays". Areas leading to the bays resemble "deltas" although, unlike geographical river deltas, they do not contain "rivers" of convergence.

Relative size can be used to describe the amount of magnification. On my monitor, the base set (X = -2.0 to +0.8, Y = -1.2 to +1.2) is about 9.5 inches wide. A small zoom (X = -0.18 to -0.14, Y = +1.02 to +1.04) results in a base width of 63 feet. Serious Mandelbrot explorers often use expansions which would fill the orbit of Mars!

To produce a graphic image of a Mandelbrot set, maximum and minimum coordinates are specified. These limits are divided by the computer screen resolution to produce a set of discrete X and Y values. Each of these points is fed into the iterative equation, which blows up or converges. The result is then plotted on the computer screen.

The resulting images are impressive in black and white, but adding color can transform them into objects of amazing beauty. The trick is to assign colors based on the number of iterations required to determine if the point is inside or outside the Mandelbrot set. This produces a plot like a contour map, where differences in elevation are represented by different colors.

Into the Depths

The primary controls for "MSet", the program accompanying this article, are the minimum and maximum X-Y coordinates, the blow-up limit, and the convergence limit. The X-Y coordinates determine the location and magnification of the set to be calculated. If the output of the equation exceeds the blow-up limit ("max", which is set to 4.0 in MSet), it is safe to assume that the output is increasing toward infinity. The number of iterations before the blow-up limit is reached is usually called "depth". The convergence limit ("crunch") is more arbitrary. It is the number of iterations after which the equation is assumed to have converged.

Some Mandelbrot set programs boast blazing speed by using an unrealistically low crunch value, such as 30. For extremely low magnifications, this can produce acceptable results. As the X-Y range is decreased, the equation tends to blow up more slowly, so a low crunch value will produce many false convergence indications.

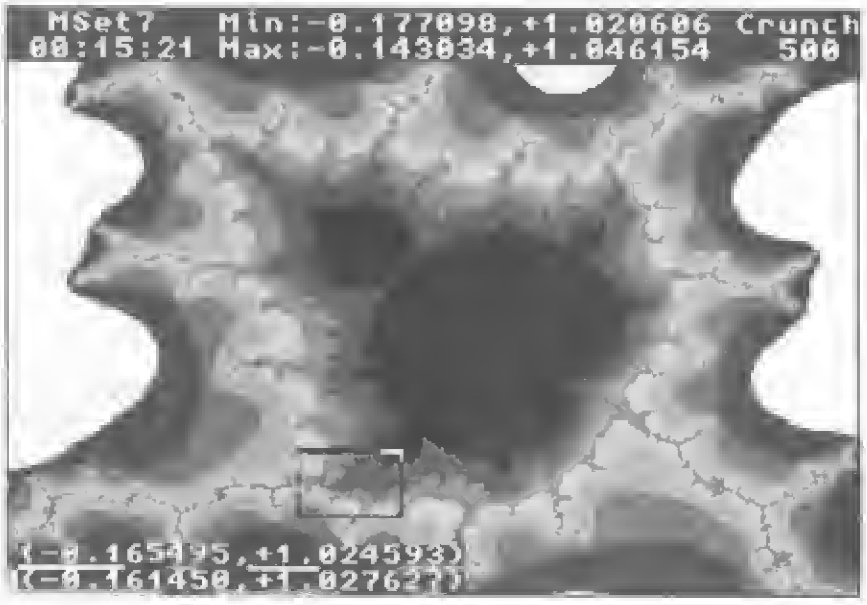
The "AnalyzeRange()" procedure in MSet shows the distribution of blow-up values. For the base set (X = -2.0 to +0.8, Y = -1.2 to +1.2), almost all blow-ups occur within 30 iterations. For a relatively small zoom (X = -0.168 to -0.152, Y = +1.028 to +1.040), 59 percent of the blow-ups occur between 30 and 499 iterations.

The default crunch value of 500 is a reasonable compromise between speed and accuracy. For high magnifications, larger crunch values should be selected.

The Base "MSet"

The program "MSet" is written in Benchmark Modula-2. It consists of several modules, with each module implementing a set of related functions. Before examining the individual modules, let's discuss a few overall concepts.

The only display mode MSet supports is low resolution (320 X 200). This provides the maximum number of colors (32), eliminates the complication of writing code to support



Top, Figure 2C. Right, Figure 2D.

multiple resolutions, calculates a specific set in minimum time, and minimizes the amount of memory and disk space required to store depth information.

The accuracy of the program is limited, because REAL numbers are used for all the calculations. REAL numbers use the Motorola Fast Floating Point format, which only gives nine digits of precision. That isn't too good for serious Mandelbrot set exploration, which often uses hundred-digit numbers. The Benchmark "System" module contains procedures for operations on LONGREAL numbers, which offer 36 digits of precision, but the Benchmark manual warns that the "System" procedures are for internal use only. It's true. If you call any of these procedures, or even attempt to perform an operation on a LONGREAL, the computer will crash.

It would be possible, but messy, to rewrite MSet to use two REAL numbers to represent each coordinate. A different language could be used; both C and BASIC support double-precision floating point numbers. Since I prefer Modula-2, the ideal solution would be to discover how to use LONGREAL numbers with Benchmark. If anyone has found a way to do this, I'd really like to hear about it.

These times were obtained by running the programs on a stock Amiga 1000. When I ran the same MSet variation on my 25 MHz Amiga 3000, the computation time dropped to 28 minutes, a speed increase of about 4.5. I actually expected a greater speed increase. The difference in clock speed alone should decrease calculation time by a factor of 3.5. Does the Modula-2 floating point library actually use the A3000's 68882 numeric coprocessor? If not, how do you access the 68882?

Zooming In

The "MSetData" module contains the control variables and constants needed by the other modules. In the "MSetDepth" module, the depth of each point is stored in a CARDINAL variable. This gives a maximum depth of 65535 and uses two bytes per point. The depth information for a low-resolution (320X200) set requires approximately 128K of memory. In Benchmark Modula-2 the maximum size of an array is 32K. In fact, 32K is the maximum space allowed for *all* variables in a single module or procedure.

If the proportions of the set's X and Y limits don't match the computer screen, the display will be distorted.

Mandelbrot sets are transformed from mathematical phenomena to objects of art by using colors to represent the depth of each point. "Depth" is the number of iterations required to determine if a point is inside or outside the set. When depth calculations for a single set require hours, even on an accelerated Amiga, it becomes very desirable to avoid recalculating the same set. Depth data is stored in the "MSetDepth" module.

The main artistic controls over a specific Mandelbrot set are the color registers assigned to specific depth values ("color selection") and the actual RGB values for each color register ("palette"). MSet has been written to give maximum control over these parameters without requiring any additional programming.

In his first *Amazing Computing* article, Paul Castonguay lists the speeds of several versions of a Mandelbrot set program. The Lattice C program took 4 hours, 42 minutes to calculate the set, using the standard IEEE floating point library. Using the less accurate fast floating point library reduced the time to 2 hours, 27 minutes.

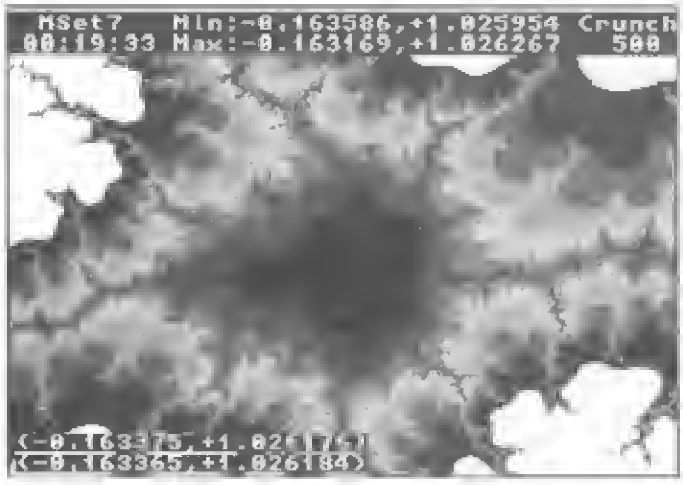
A version of MSet calculated the same set in 2 hours, 12 minutes. Since MSet uses a slightly different algorithm from the C program, the times can't be compared directly. It's safe to say there is no significant difference in execution speed between C and Modula-2 programs for this type of computation. The floating point library is the most important factor in determining Mandelbrot set calculation speed.

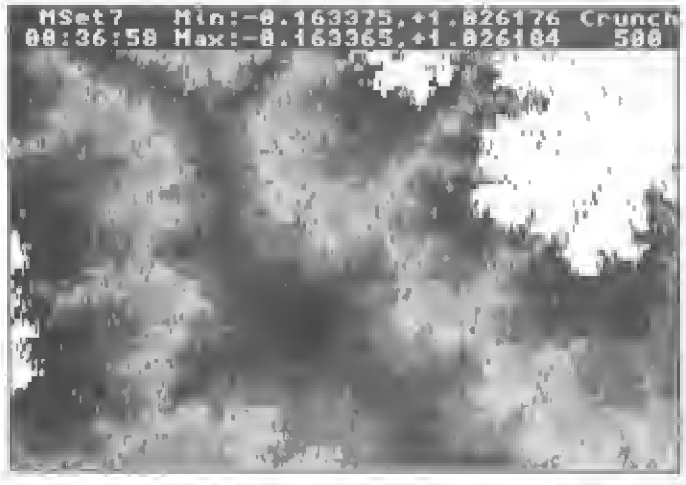
Dynamic memory allocation to the rescue! Memory obtained using the "AllocMem()" procedure does not occupy variable space and is not subject to the 32K limit. MSetDepth contains procedures for allocating and releasing the memory where depth information will be stored, as well as storing and retrieving the depth of a single point. The module can also save and read the depth data from disk.

Most of the Amiga-specific code is in "MSetResources". This module allocates and releases system resources (display screen, window and depth storage), clears the screen, plots points, displays the set limits at the top of the screen, draws a box showing newly-selected limits, sets color register RGB values, and performs a few utility functions.

Isolating machine-specific code in a single module is a good technique for writing programs which can be easily ported to another system. It also reduces the possibility of side effects. MSetResources is the only module with access to sensitive system data structures, such as the screen and window pointers. No other module can accidentally modify system data.

The color selection and palette procedures are in "MSetColors". "AddColor()" is used to build a linked list of color range records. Each range record contains low and high depth values and a color register number. The "Color()" procedure scans the list for a depth range matching the input depth and returns the appropriate color register number. If no match is found, the highest color register number (31) is returned. "ReleaseColors()" frees the memory used by the





Top to bottom: Figures 2E-2G. Each zoom reveals more intricate detail and fascinating patterns.

current color range list. It is used before a new color selection list is entered and at program termination.

MSetColors contains three procedures which create color selection lists. These can be used as models for your own color selection algorithms. "AC3ColorsSetUp()" is a 32-color version of the 16-color selection method in Paul Castonguay's third Mandelbrot set article. This gives good results for a wide variety of sets and is the program's default color selection method.

I devised the other two algorithms in MSetColors. "EqualPointsColorsSetUp()" distributes the color selection so that each color is assigned to an equal number of points. When a large number of points have the same depth, a single color is assigned to that depth. Otherwise, one color is assigned to a narrow range of depths. This selection method must be run after a new set is calculated or after the depth data of an old set is loaded from disk. It always produces a good-looking display.

"EqualRangeColorsSetUp()" divides depth values from 0 to crunch into equal color ranges. This is a very logical and mathematically precise color selection method which produces a boring and almost useless display. I left it in the program to show the limits of using logic and strict mathematical relationships when exploring Mandelbrot sets. Intuition, inspiration and experimentation will take you much further.

There are two procedures to set sample palettes in MSetColors. "SetRainbowPalette()" offers 11 colors, with each color having several intensity levels. Except for black, the colors are arranged so that brightness increases with depth. The specific colors were chosen for purely aesthetic reasons. Displays which use this palette tend to have a brilliant, jewel-like quality.

The colors in "SetDepthPalette()" are arranged so brightness decreases as depth increases. This produces a display a little like a contour map of the ocean floor, where deeper blues indicate deeper water. The "depth" palette still uses multiple colors. Since a standard Amiga only has 16 levels (four bits) of red, green and blue color values, it's impossible to get 32 shades of a single color. If we restrict the display to 16 colors, it's difficult to see the difference between adjacent color values. Sigh! I wish Commodore would hurry up and increase the number of standard Amiga colors.

The "MSetAnalyze" module contains two depth analysis procedures. "AnalyzeDepth()" displays the number of occurrences of each depth value, sorted from most to least occurrences. It uses a "comb" sort from the April 1991 issue of *Byte*. By a happy coincidence, this algorithm was developed on an Amiga 2000. It can help you decide how to assign depth ranges.

"AnalyzeRange()" produces a percentage graph of depth values in a set of ranges. It helps determine if a sufficiently large crunch value is being used. If depth values are concentrated in the low end or middle of the depth range, the crunch value is probably adequate. If a large percentage of depth values occupy the high end of the scale, the crunch factor should be increased for accurate computation.

"MSetFiles" handles all disk input and output. All the procedures in this module prompt for a file name, open the file, read or write the data, and close the file.

"LoadColorAlloc()" uses the procedures in MSetColors to create a color selection list. It reads a text file with one color range specification per line. Each line contains a low depth value, a high depth value and a color register number. The numbers can be separated by any non-numeric character(s) except a semicolon (";"), which indicates the rest of the line is a comment.

"LoadPalette()" reads a similar text file. Each line contains a color register number (0-31), followed by red, green and blue color values (0-15).

"MSetLoaded()" reads Mandelbrot set control data and copies it to the global variables in MSetData. It also reads a file containing the depth values and stores them in the depth array. "SaveMSet()" writes the control and depth data to disk. All calculation data and results are saved and can be restored, making it possible to interrupt a computation, save it to disk, and restart it at a later time.

Every type of MSet data file has a unique suffix: ".msdata" for control data, ".msdepth" for depth information, ".mspalette" for palette specifications and ".msalloc" for color allocations. All the disk operations prompt for a file name, append the appropriate suffix, and read or write the file(s).

"MSet" is the main module. It contains the calculation routines and the program control logic.

"CalculatePoint()" determines if a single coordinate pair (input as "i" and "j") are inside or outside the Mandelbrot set. The "i" and "j" inputs are applied to the equations "Xnew = x * x - y * y + i" and "Ynew = 2 * x * y + j". The new "x" and "y" values are substituted into the equations and they are recalculated until the "x" and "y" values blow up or the number of iterations exceeds the "crunch" value. The iteration count when a termination condition is reached is returned as "depth".

"CalculateMSet()" controls the calculation of a complete Mandelbrot set. If the flag variable "newCalculation" is true, the control variables, display and depth array are initialized and a new calculation is started. Otherwise, an interrupted calculation is continued. For each point, the procedure computes coordinates, obtains depth from "CalculatePoint()", determines the display color associated with that depth, plots the depth on the screen, and stores the depth value. If a keypress is detected after a point has been processed, we escape from the procedure and fall back into the command processing loop.

If the proportions of the set's X and Y limits don't match the computer screen, the display will be distorted. After the limits are entered, "AdjustAspectRatio()" modifies the vertical limits to be 3/4 of the horizontal limits, assuring a correct aspect ratio.

"RedisplayPic()" uses the data from the depth array to redraw the entire screen and displays the calculation parameters at the top of the screen.

The rest of the procedures are all menu processors. They display a set of options, obtain a command character from "GetChar()" and call the appropriate procedure.

Set The Controls For The Heart Of The Sun

Table 1 outlines MSet's controls. To avoid complicating the program with Intuition code, a simple character-based interface is used. If the program is run from a console window, like the window created by the "Run Main Module" function of the Benchmark editor, it is controlled by single keypress commands. If MSet is run from the Shell or CLI, the "Return" key must be pressed to send keystrokes to the program. Since commands are entered into a Workbench window and graphic output occurs on a custom screen, it is necessary to use the Left-Amiga-m key (the Amiga "Toggle Screen" function) to switch between control and display screens.

Table One

MSet

```
A - Analyze
    D - Depth
    R - Range
C - Calculate MSet
D - Disk Operations
    A - Load Color Selection
    L - Load MSet
    P - Load Palette
    S - Save MSet
K - Color Palette
    D - Depth Palette
    R - Rainbow Palette
N - Set Up New Calculation
O - Restore Old Parameters
P - Enter Parameters
    x - Xmin
    y - Ymin
    X - Xmax
    Y - Ymax
    C - Crunch
    A - Accept
    Q - Quit
Q - Quit
R - Redraw Current MSet
S - Color Selection
    A - Amazing Computing 3
    N - Set Number of Colors
    P - Equal Point
    R - Equal Range

Command?
```

Most of the commands invoke procedures which I have already discussed. These should be self-explanatory. I'm going to describe a few of the less obvious control functions and demonstrate how some of the MSet features are used.

"Enter Parameters" is used to enter or modify X and Y limits and the crunch value. If the parameters are accepted,

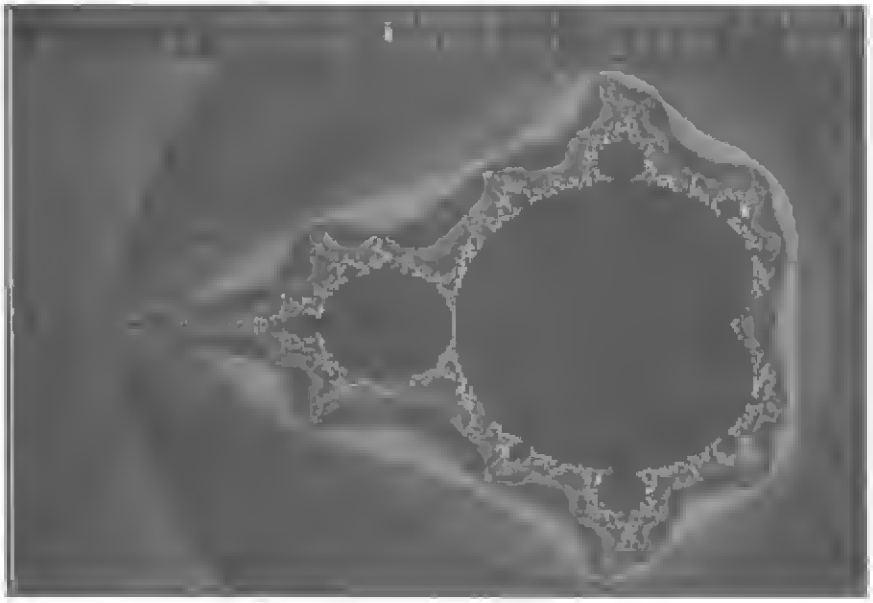
the new coordinates are drawn on the display screen. If these coordinates will fit within the current display, the "limit rectangle", which shows the new limits, is also drawn (Figures 2A thru 2E).

"Restore Old Parameters" resets the control variables to the previous limits and crunch value.

"Calculate MSet" starts a calculation, using the current control parameters. If the calculation is partially completed, it will be continued, starting from the point where it was interrupted.

"Set Up New Calculation" resets the control variables to start the calculation from the beginning.

"Redraw Current MSet" redraws the screen using the current depth data. This is mainly used to clear the limit box from the display screen.



The only display mode MSet supports is low resolution (300x200). This provides the maximum number of colors and eliminates the complication of writing code to support multiple resolutions.

"Set Number of Colors" controls the number of colors used in the algorithms which select colors to associate with depths.

Shades Of Gray

The illustrations were created using MSet7, an advanced version of the MSet program, because it displays the calculation time for a set and can save a screen as an IFF file. See the note at the end of this article for more information about MSet7.

The Amiga is a color computer, but *AC's TECH* is a black-and-white publication. To avoid the problems of reproducing intricately detailed 32-color pictures in this format, I converted the illustrations to eight shades of gray.

Reducing the number of colors produced an unexpected benefit. Removing extraneous detail made the overall structure of the areas of varying depth more apparent.

Let's go over one of the conversions. I started by loading "Base Set" (Figure 1), which had been calculated earlier and saved

The first conversion step was to reduce the number of colors. I selected eight colors with the "Set Number of Colors" function and picked the "Equal Point" algorithm from the "Color Selection" menu. This produced an eight-color display with the various zones in the set clearly delimited.

I saved the display screen as an IFF file and loaded the picture into Deluxe Paint IV to experiment with the colors. I could have used the "Load Palette" function and my text editor to adjust the colors, but Deluxe Paint was much faster. I chose alternating light and dark levels of gray to highlight the boundaries between the zones and to make the headings (color 1 over color 0) stand out.

When I had found a satisfactory color set, I entered the red, green and blue values from the Deluxe Paint color requester into a text file (Table 2). Then I used "Load Palette" to set the MSet colors. This palette was used for all the illustrations.

"Load Color Selection" could have been used instead of "Equal Point". This would have been desirable if I had wanted the same colors assigned to the same depth ranges in each image, rather than the best fit of colors to depths for each individual image. Table 3 shows a color selection file optimized for the base set.

Nothing But Zooms

Figures 2A thru 2F demonstrate the general procedure for finding interesting areas of the set. Each image is a close up of the rectangular area shown in the previous picture. Some of the details in the original 32 color screens aren't visible in the illustrations.

The process is pretty simple: examine a low-magnification view of the set for something that looks interesting and zoom in on it. The limit rectangle (Figures 2A thru 2E) will show the exact site of your next zoom. Don't bother zooming into an area that's all lake. That will take a lot of computer time and reward you with a completely black screen.

That's enough theory. Let's do some zooms.

The area around the north bay of the base set (Figure 2A) contains a lot of interesting detail, including a black dot. Areas containing black dots or blobs are good candidates for zooms.

Figure 2B shows the first zoom. Some of the detailed structure of the "river deltas" around the bay is revealed and the black dot has expanded into an irregular oval.

A zoom into the area surrounding the oval produces Figure 2C. A version of the base set, rotated 45 degrees, is now visible. Figure 2D is a close-up of the newly-revealed lake. At this magnification, the edges of the lake are getting a bit fuzzy. Many of the points near the lake have coordinates which will eventually blow up, but the calculations are cut off by the crunch limit of 500. For consistency, I used a crunch value of 500 for all the illustrations in this article. For greater accuracy, crunch should be increased for higher magnifications.

I decided to do the next zoom below the southwest bay because I saw another black dot in this area. Figure 2E is the result. The black dot in the center has expanded into a cubistic version of the base set and several more black blobs are visible. Notice that the sub-bay in the upper right corner has grown another sub-bay of its own.

Zooming in on the black blob in the center produces Figure 2F. We've returned to the base set, except the size of this region is approximately one ten-thousandth of the original. At this level of magnification, the base image would be 6075 feet (1.15 miles) across.

This is the region where things start getting really interesting. It's also the point where we hit the wall. This last image shows a certain blockiness that can't be eliminated by increasing the crunch value. Any further zoom will produce images that look like they were constructed out of blocks (Figure 2G). We have reached the accuracy limit of the single-precision floating point REAL numbers that MSet uses.

Convergence

Even with its current magnification limits, MSet offers plenty of surprises. Zooming in on the other black blobs in Figure 2E revealed base images, rotated into orientations I've never seen before. The distinctive heart and circles of the base image aren't the only figures present in the Mandelbrot set, but I'll leave the others for you to discover.

If you want to explore outside the "confines" of the classic Mandelbrot set, other equations can be used in the calculation procedure. Paul Castonguay's fifth article gives two examples of interesting alternate equations.

Three highly technical books on fractals are Benoit Mandelbrot's own *The Fractal Geometry of Nature* (W. H. Freeman, 1982), *The Beauty of Fractals* (Springer-Verlag, 1986) and *The Science of Fractal Images* (Springer-Verlag, 1988), both by H-O. Peitgen and Dietmar Saupe. A somewhat less challenging work is *The Armchair Universe* by A. K. Dewdney (W. H. Freeman, 1988), author of the *Scientific American* "Computer Recreations" article that introduced Mandelbrot sets to the general public (August 1985, pages 16-25).

The best non-technical introduction to fractals and Mandelbrot sets I have seen is Paul Castonguay's "Fractal Fundamentals" series in *Amazing Computing* (March, July, and October 1989; January and June 1990; April 1991).

There are also Mandelbrot set video tapes, such as "Nothing But Zooms" (Art Matrix, P.O. Box 880, Ithaca NY 14851) and "A Fractal Ballet" (The Fractal Stuff Company, P.O. Box 5202, Spokane WA 99205-5202).

Happy hunting. See you at the lake.

A version of "MSet" with an Intuition interface and numerous additional features is available from the author. Full source code and an executable program are included. Send a floppy disk in a self-addressed mailer with return postage to "Jim Ollinger, 5225 Marymont Drive, Austin, Texas 78723".

8 Level Gray Scale Color Allocation

low depth,	high depth,	color
000,	001,	00
002,	003,	01
004,	004,	02
005,	005,	03
006,	007,	04
008,	013,	05
014,	499,	06
500,	500,	07

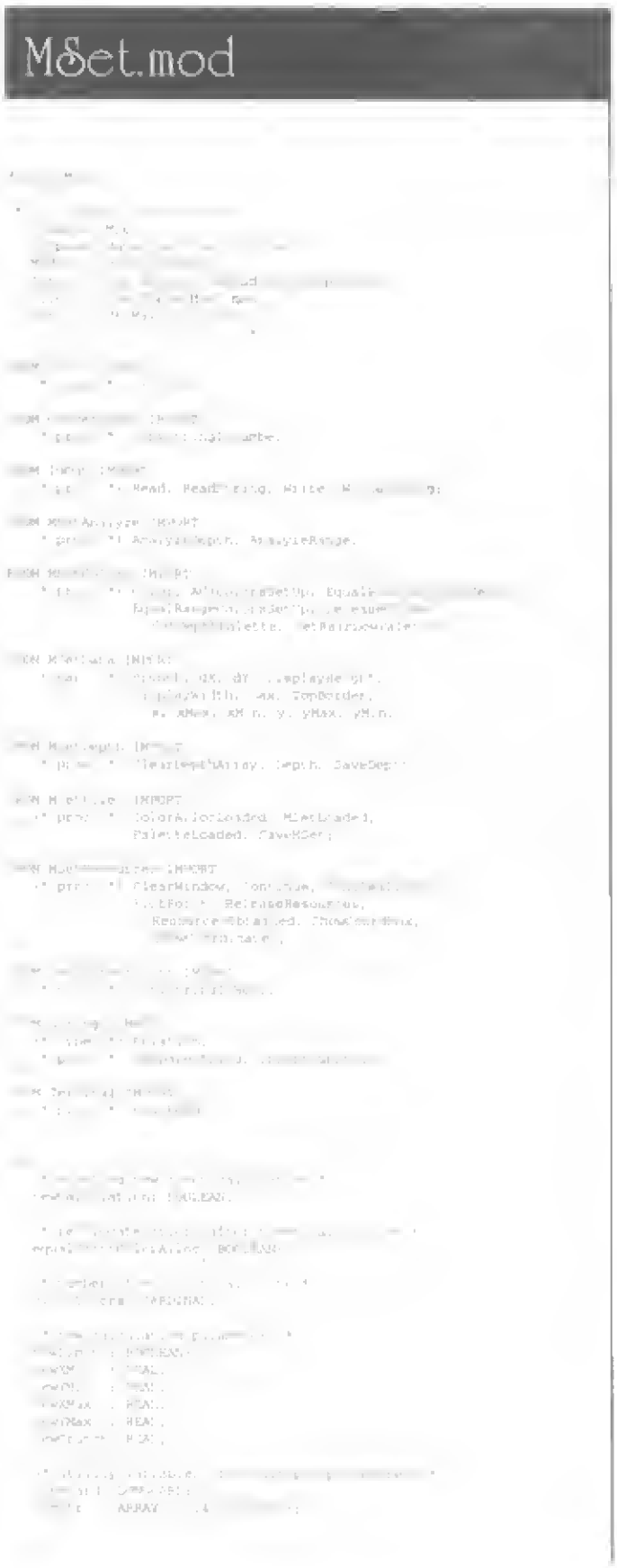
Table Two

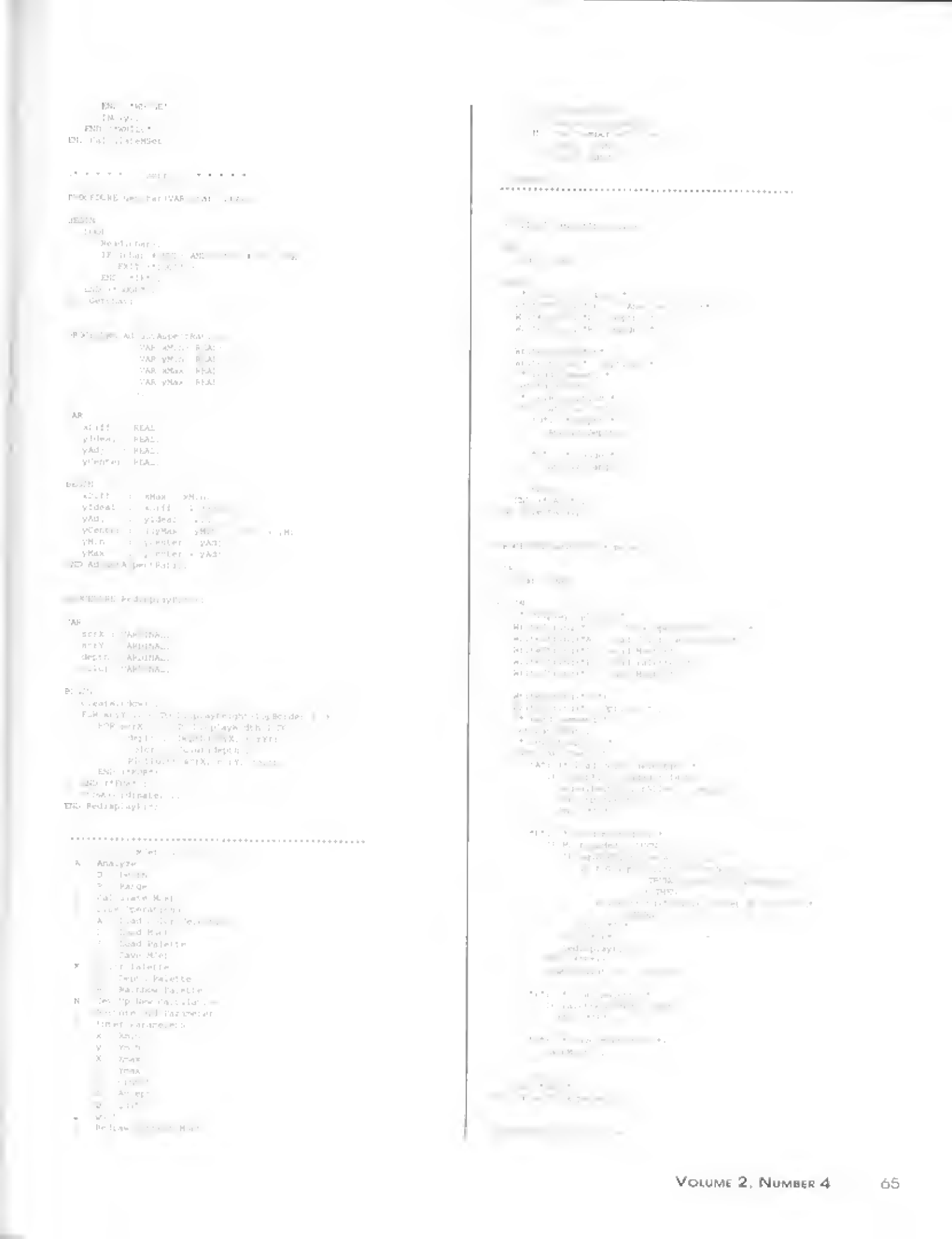
8 Level Gray Scale Palette

color	red	green	blue	description
00	15	15	15	White
01	03	03	03	Gray 1
02	07	07	07	Gray 3
03	11	11	11	Gray 5
04	13	13	13	Gray 6
05	09	09	09	Gray 4
06	05	05	05	Gray 2
07	00	00	00	Black

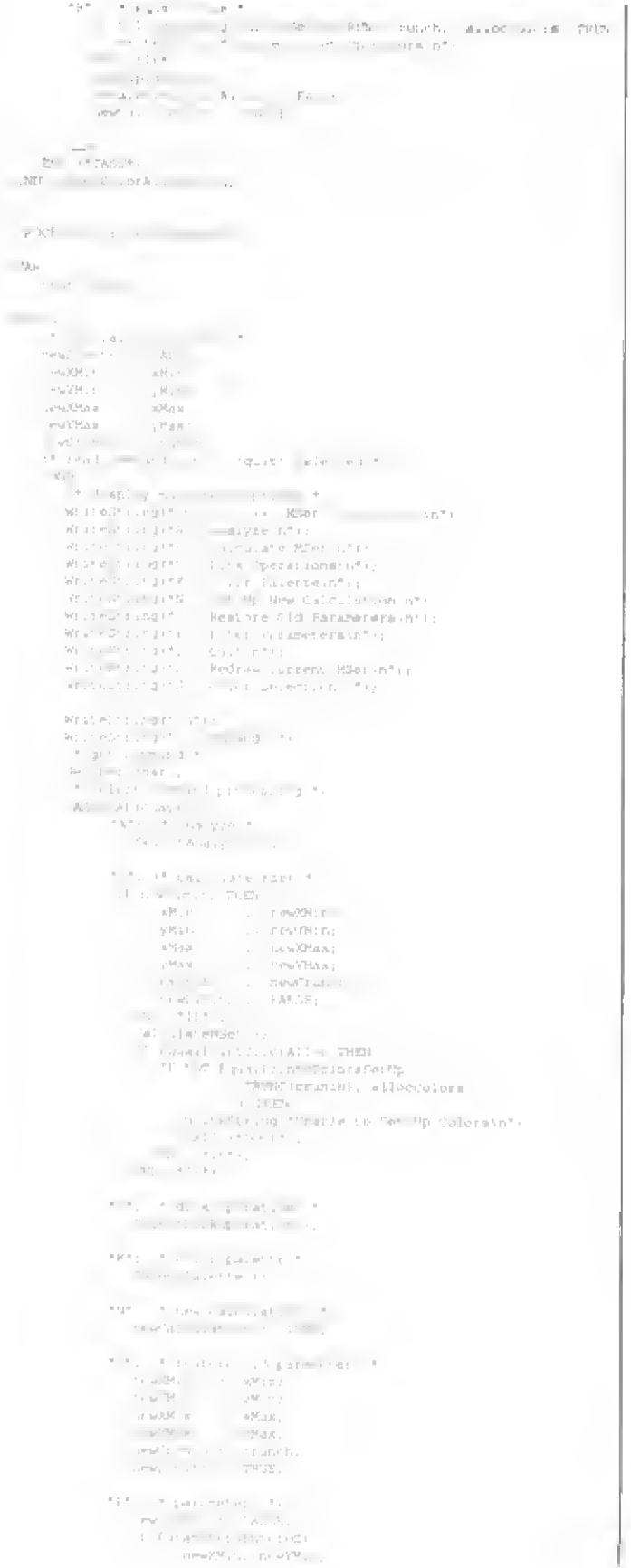
Table Three

MSet.mod





```
VAR
   char: CHAR;

BEGIN
   (* display options *)
   WriteString("---  Palette  ---"n");
   WriteString("D   Depth Palette"n");
   WriteString("R   Rainbow Palette"n");

   WriteString("?");
   WriteString("   Option) ");
   (* read command *)
   GetChar(char);
   (* select option *)
   CASE CAP(char) OF
      "D": (* Depth palette *)
         SetDepthPalette();

      "R": (* rainbow palette *)
         SetRainbowPalette();

      ELSE
   END (*CASE*);
END SelectPalette;


PROCEDURE ParametersEntered(
             VAR xMin: REAL;
             VAR yMin: REAL;
             VAR xMax: REAL;
             VAR yMax: REAL;
             VAR crunch: REAL
            ): BOOLEAN;

VAR
   newXMin  : REAL;
   newYMin  : REAL;
   newXMax  : REAL;
   newYMax  : REAL;
   newCrunch: REAL;
   char     : CHAR;

BEGIN
    (* store new parameters *)
   newXMin   := xMin;
   newYMin   := yMin;
   newXMax   := xMax;
   newYMax   := yMax;
   newCrunch := crunch;

   (* get new parameters *)
   LOOP
       (* display parameters *)
      WriteString("\n=== Parameters ===\n");
      WriteString(" x: X min  = ");
      ConvRealToStr(newXMin, numStr, 8);
      WriteString(numStr);
      WriteString("\n");
      WriteString(" y: Y min  = ");
      ConvRealToStr(newYMin, numStr, 8);
      WriteString(numStr);
      WriteString("\n");
      WriteString(" X: X max  = ");
      ConvRealToStr(newXMax, numStr, 8);
      WriteString(numStr);
      WriteString("\n");
      WriteString(" Y: Y max  = ");
      ConvRealToStr(newYMax, numStr, 8);
      WriteString(numStr);
      WriteString("\n");
      WriteString(" C: Crunch = ");
      ConvRealToStr(newCrunch, numStr, 8);
      WriteString(numStr); '
      WriteString("\n");
      WriteString("\n");
      WriteString(" A: Accept\n");
      WriteString(" Q: Quit\n");
      WriteString("\n");
      WriteString("Command: ");
      (* process commands *)
      GetChar(char);
      WriteString("\n");
      CASE char OF
         "x":
            WriteString(" X min: ");
            ReadString(numStr);
            IF CompareString(numStr, "") # equal THEN
               newXMin := ConvStringToReal(numStr);
            END (*IF*);

         "y":
            WriteString(" Y min: ");
            ReadString(numStr);
            IF CompareString(numStr, "") # equal THEN
               newYMin := ConvStringToReal(numStr);
            END (*IF*);

         "X":
            WriteString(" X max ? ");
            ReadString(numStr);
            IF CompareString(numStr, "") # equal THEN
               newXMax := ConvStringToReal(numStr);
            END (*IF*);

         "Y":
            WriteString(" Y max ? ");
            ReadString(numStr);
            IF CompareString(numStr, "") # equal THEN
               newYMax := ConvStringToReal(numStr);
            END (*IF*);

         "C", "c":
            WriteString(" Crunch ? ");
            ReadString(numStr);
            IF CompareString(numStr, "") # equal THEN
               newCrunch := ConvStringToReal(numStr);
            END (*IF*);

         "A", "a":
               xMin   := newXMin;
               yMin   := newYMin;
               xMax   := newXMax;
               yMax   := newYMax;
               crunch := newCrunch;
               RETURN TRUE;

         "Q", "q":
               RETURN FALSE;

         ELSE
      END (*CASE*);
   END (*LOOP*);
END ParametersEntered;


PROCEDURE SelectColorAllocation();

VAR
   char: CHAR;

BEGIN
   (* display options *)
   WriteString("\n ==  Color Selection  ==\n");
   WriteString("A   Amazing Computing 3.in");
   WriteString("N - Set Number of Colors\n");
   WriteString("P   Equal Point\n");
   WriteString("R   Equal Range\n");

   WriteString("\n");
   WriteString("   Option) ");
   (* read command *)
   GetChar(char);
   (* select option *)
   CASE CAP(char) OF
      "A": (* amazing computing *)
         IF NOT ACTColorsSetUp(TRUNC(crunch), allocColors) THEN
            WriteString("Unable to Set Up Colors\n");
         END (*IF*);
         RedisplayPicture;
         equalPointColorsAlloc := FALSE;
         newCalculation := FALSE;

      "N": (* number of colors *)
         WriteString("Number of Colors to Select? ");
         ReadString(numStr);
         IF CompareString(numStr, "") # equal THEN
            IF ConvStringToNumber(numStr, longCard, FALSE, 1, ) THEN
               allocColors := longCard;
            END (*IF*);
         END (*IF*);

      "P": (* equal point *)
         IF NOT EqualPointsColorsSetUp(TRUNC(crunch), allocColors) THEN
            WriteString("Unable to Set Up Colors\n");
         END (*IF*);
         RedisplayPicture;
         equalPointColorsAlloc := TRUE;
         newCalculation := FALSE;
```



All the source code and executable files for The Joy of Sets can be found on the AC's TECH Disk.

Please write to:
Jim Olinger
c/o ACs TECH
P.O. Box 2140
Fall River, MA 02722-2140

Advanced Scripting

A Login Utility

by Douglas Thain

The new features brought to the shell by System 2.0 make AmigaDOS a unique language for string and file manipulation. With two new commands written in C in our arsenal, we can create a login utility. This security device will prevent unwanted access to your machine and will allow each user of your system to have a personalized environment to work in.

TWO NEW COMMANDS

code and rawread

Listings One and Two are programs that fill in for commands AmigaDOS doesn't have. They were compiled using Matt Dillon's *DICE*, but any Amiga C compiler should be sufficient. Compile using the residentiable switch:

```
1> dcc -r rawread.c -o rawread
1> dcc -r code.c -o code
```

Place them in C: or anywhere in your search path—making them resident is even better.

rawread reads in a single line from the console and can be set to echo what is typed, a dummy character, or nothing at all. Although the command set can usually be used to read in data, it does not work well from the RAW: console that our script will be running in. **rawread** will allow us to hide what is being typed when the user enters his/her password. **rawread** is executed with one argument—ECHO, MASK, or BLIND. These switches will echo what is being typed, echo "#"s, or echo nothing, respectively.

code takes a string as an argument, and changes it into a very large number. We will use **code** to encrypt each user's password. **code** is nearly non-reversible; it is next to impossible to determine a password that matches the value that is output.

NEW SCRIPTING TECHNIQUES

Variables and $

Local variables are a new feature with AmigaDOS 2.0. Previously, environmental variables were stored in the ENV: directory, and manipulated with getenv and setenv. The problem with this method was that processes shared the same data, and scripts had to go to great lengths to ensure that their variables had unique names. ENV: variables are still supported in the same manner, but local variables remove much of the hassle while keeping data from prying eyes.

Local variables are manipulated using the commands **get** and **set**. From the shell, enter set. You should see something like this:

```
process     5
RC          0
Result2     0
```

and possibly others that other programs have set up.

process, **RC**, and **Result2** are variables maintained by the shell; they are always present. **process** contains the same process number as shown by the command status. RC is the return code of the last command—0 is success, 5 is warn, 10 is error, 20 is fail. Result2 contains the number of a special message that the command why uses to explain errors. These values are now available for the user to parse, instead of relying on the existing AmigaDOS commands such as why and fault.

To set a variable, simply enter set with the variable name and its value:

```
1> set myname foo
1> set val 4
1> set
   myname      foo
   process     5
   RC          0
   Result2     0
   val         4
```

To retrieve a variable, use the command **get**.

```
1> get myname
foo
```

in AmigaDOS 2.0

However, the use of **get** is limited. The $ operator is much more useful. To use a variable as an argument, enter $ followed by the variable name:

```
1> echo $myname
foo
1> rename test $myname
```

To completely remove a variable, use the command **unset**.

```
1> set
    process     4
    RC          0
    Result2     0
    val         4
1> unset val
1> set
    process     4
    RC          0
    Result2     0
```

Backquoles

The backquote (`) (the key above tab and left of 1) is probably the most powerful part of AmigaDOS 2.0. A string in backquotes on a command line is executed before the main command, and the backquoted string is replaced by its own output. An example:

```
1> break `status command=blanker`
```

The shell executes the command **status command=blanker** first. This command returns the process number of the currently running program blanker—let's say it's process 6. 'status command=blanker' is replaced with "6", and the command executed is really this:

```
1> break 6
```

This little gem sends a break signal to any named program without previous work by the user.

This sort of command will come in very handy in our login script:

```
1> set num `code password`
```

code password is executed first, and outputs an ugly number. The number is inserted into the command, and it becomes

```
1> set num 12341234
```

The backquote method eliminates the need for schemes that rely on PIPE:, and makes AmigaDOS code much more pleasant to create and easier to read.

Redirection and *

Data redirection is not new to AmigaDOS 2.0, but deserves a quick review. When a program is executed from the CLI, it is given two data streams: one to ouput to, and one to input from. In C, these streams are better known as **stdin** and **stdout**. Both streams usually point to the window from which the command was run. However, these streams can be made to point to other files by using the greater than (>) and less than (<) symbols to specify files for output and input, respectively. For example, the command

```
1> echo >t:temp "Testing"
```

would not result in any text to be displayed. The input file stream has remained on the window where it usually is, but the ouput stream has pointed to t:temp. If you examined the file t:temp, you would find that it contained "Testing".

The asterisk (*) character, when used with redirection, designates the current window. It is rarely necessary to specify the current window for input or output (it is implicitly established,) but the command **rawread** when used with backquotes presents a special problem. You would expect the command

```
1> set logname `rawread BLIND`
```

to execute **rawread**, accept input, then use it as the second argument. However, it turns out that when using backquotes, no input stream is created for the inner command. So, in order to read from the current window, we must explicitly specify it:

```
1> set logname `rawread <* BLIND`
```

This command will give **rawread** the input handle it needs. Without it, the command would hang forever, waiting for input.

A Note About Quotes

A complication that can occur unexpectedly when using variables and backquote piping is that a string might contain a space in the middle. Consider this case:

```
1> set
    foo         key board
    process     4
    RC          0
    Result2     0
1> echo $foo
    argument invalid or line too long
```

What was actually executed was:

```
1> echo key board
```

The command **echo** won't accept more than one argument. If you were typing the command directly, you would execute

```
1> echo "key board"
```

The same thing can be done with variables and backquote pipes:

```
1> echo "$foo"
1> set test "`rawread <* ECHO`"
```

These force the output into one argument, ensuring successful execution. A good rule of thumb is to always enclose variables and backquoted commands in double quotes. Even if you expect the input to be continuous, put quotes around it so you won't be surprised later on. Of course, if you *intend* to pass more than one argument to a command, it is perfectly legal to leave the quotes out.

THE SYSTEM LOGIN SCRIPT

Directory Setup

On your hard drive or boot disk, you will need a directory to contain all the user directories which will contain each user's personal files and scripts. This directory must have USERS: assigned to it—do this in s:startup-sequence along with your other assignments.

Installing the Script

Listing Three is the login script. This goes in your S: directory as **system-login**. Once you have your user directories set up as you like them, test the script by executing

```
1> newshell  RAW:40/50/220/100/Shell  from
s:system-login
```

The login window will pop up, showing a message, the time, and a "login:" prompt. Enter your login name (the same as your home directory) and your password, and a newshell will be started with your user-login file. To logout, just end all the programs you have started, and **endcli** your shell. If you wish to end the login script, enter "exit" at the "login:" prompt, and the window will close.

If you simply want to run this from the shell once your system is booted, an alias for the above command is all that is necessary. However, if restricting access is your goal, you'll want the startup-sequence to start system-login so that it is the only window on the screen once your machine has booted up. This involves one last trick so we can get around the program **iprefs**. iprefs is usually run from the startup-sequence to set up your preferences. If iprefs needs to change something drastic, like the size or resolution of the workbench screen, it pops up a requester asking you to close all windows that aren't drawers. If the login window appears before the startup window closes, iprefs will never have a chance to change the screen to your settings. To fix this, we need one short script that is called from startup-sequence that waits long enough for the startup window to close so iprefs can seize the moment and adjust the screen. The script can then call our above newshell command, and exit.

This script (Listing Four) is called **startup-wait**. With startup-wait and system-login in the s: directory, the last step is to add

```
run >NIL: execute >NIL: s:startup-wait
```

to the last line before endcli in your startup-sequence.

When you reboot, iprefs may pop up a requester, but just wait. The startup window will disappear, iprefs will reset the screen, and after a few moments, the login prompt will pop up.

Eventually, you may wish to remove any **newshells** or **loadwbs** from startup-sequence so that system-login is the only thing on your screen when you boot up. However, the first time, leave something else running so that you'll have a backup in case the script fails.

Hints

Do *lots* of dry runs before you install the scripts in your startup-sequence! There's nothing worse than getting locked out of your system disk.

Temporarily eliminate the **failat** command at the beginning of system-login so that any serious errors will terminate the script instead of locking you out.

When working at the RAW: console outside of the login script, you cannot see what you type, and the return key does not function. Enter your commands slowly—delete is not supported—and press Control-J instead of return.

This program will not keep users who already have access from messing with other user's files (at least not until AmigaDOS 3.0 arrives). However, if you write user-login scripts carefully, you can give some users a text-menu driven interface to run a select few programs, some a personalized shell, and others the Workbench.

Use quotes around as many variables and backquote pipes as possible.

Things to Add

A logout script for each user isn't such a bad idea. Create an IconX icon or alias endcli to point to such a script.

Can you write a script to send mail between users?

Listing One

```
/*  Listing One */
/* rawread.c
   by Douglas Thain

      Reads data from 'stdin unit. linefeed or 'carriage
   return, then echoes to stdout.  Echoes data back to stdi
   as it is entered.  ECHO echoes chars typed back to input
   MASK keyword forces  echo of '*'s instead of data.  BLIND
   keyword forces no echo.  Use MASK or ECHO to read from
   RAW:  Has BLIND to read from CON:

   Compiled and debugable using DICE:
   dcc rawread.c -o rawread
*/

/* <libraries/dos.h> is instead of <stdio.h> here.  DICE won't
   allow writes to stdin */

#include <libraries/dos.h>
#include <ctype.h>

/* macros for each modes */
#define MODE_ECHO 0
#define MODE_MASK 1
#define MODE_BLIND 2

char irp[80];

main(argc,argv)
int  argc;
char *argv  ;

   int  pos,mode=MODE_ECHO;
   char ch;

    /* Well, if we can't output, what are we doing? */
   if(IsOutput()==FALSE) exit(20);

    /* Is there an argument? */
   if(argc>1) {
       /* Only check first letter of argument */
      switch (ToUpper(argv[1][0])) {
         case 'B': mode MODE_BLIND; break;
         case 'E': mode MODE_ECHO;  break;
         case 'M': mode MODE_MASK;  break;

         default:
         Write Output(),
            "Usage: rawread [ECHO|MASK|BLIND]\n",23);
         exit(5) ;
         break

      }

   while(1)
      irp[pos]=0;

       /* Fetch a char */
      ReadInput() ,&ch,1);

       /* backspace pressed */
      if(ch==8) {
         if(pos>0) {
            Write Input(),&ch,1);
            pos--;
            }
         continue;

       /* Carriage return if RAW:, or linefeed if CON: */
      if( (ch==13)  (ch==10) ) {
         WriteOutput(),irp,pos);
         exit(0);


       /* If it's a printable char, add it to the buffer
          and echo, if appropriate.  Ignore non printables */
      if(isprint(ch)) {
         if(pos<79) {
            if(mode==MODE_ECHO) WriteInput(),&ch,1);
            if mode==MODE_MASK) WriteInput(),"*",1);
            irp[pos]=ch;
            pos++;
            }
```

Listing Two

```
[illegible]
```

Listing Three

```
[illegible]
```

```
. Same method as for reading ".logname"
set logpass " rawread <* MASK *

; Set "logcode" to whatever the user set
; "code $logpass" was.
set logcode " code $logpass "

echo ""
echo ""

. Does the entered user directory exist, i.e. a file
. containing the encoded password?
. If so, set "realcode" with the contents of the f...
. "realpass".
. If "realcode" and "logcode" are the same, grant access.
. Otherwise, deny.
if exists "user:$logname/codepass"
   cat "user:$logname"
   set realcode " type codepass "
   if "$realcode" eq "$logcode"
      pkg enter
      exit
   endif

exit "Access denied."
 s p bark begin

  Currently, login begins here
lab enter
echo "Access granted."

: Clear out unneeded variables
unset logpass
unset real.code
unset logcode

. If there is a file containing the user's real name
. If so, set "realname" to its contents
if exists real.name
   set realname " type realname"
   endif

  Add additional user info in the same way

. Start a new shell of default dimensions and a title
; the contents of "realname".  The shell will begin by
. executing the user's custom script.  The new shell
. variables will be a copy of the current ones.
newshell "CMS:  "$realname" from user log.

; Start again!
exec ba > begin
```

Listing Four

```
   Story Four
  startup-wait
wait <NIL: 5 secs
new hel RAM:A3.4 /Z/Z/2/system .cg. 1 ...
end/i <NIL:
```

Listing Five

```
   ...ating Save
  Login script for a ...hell user

 ...ack 2k 20
e hc "Ec"
echo "Welcome, $logname"
echo "Time is now: " noline
date
echo "Chip. .vail chip   " noline
echo "Fast. .vail fast "
alias clear    echo "Ec"
alias logout   execute user-logout
alias ls       .cat quick:head
alias ps       status full
alias cp       copy !
alias flush    avail flush
alias lf.ush   delete t:*
alias ed       ...r * t:
prompt "  ...ec> "
```

Listing Six

```
   ...ng ...
  ...p ... a multiple...

...ec "Eg/"
date
...
...  "...
/cache
...
...
```

Listing Seven

```
   ...ng ...
  ...n script for a ...

...ack 2k 20
e hc "Ec"
...
...
echo "...
...  "...
echo "Welcome, $logname"
...  "...
echo "Time is now: " noline
date
echo "Chip. .vail chip   " noline
echo "Fast. .vail fast "
...
...
alias clear    echo "Ec"
alias logout   execute user-logout
alias ls       .cat quick:head
alias ps       status full
alias cp       copy !
alias flush    avail flush
alias lf.ush   delete t:*
alias ed       ...r * t:
prompt "  ...ec> "
```

Please write to:
Douglas Thain
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

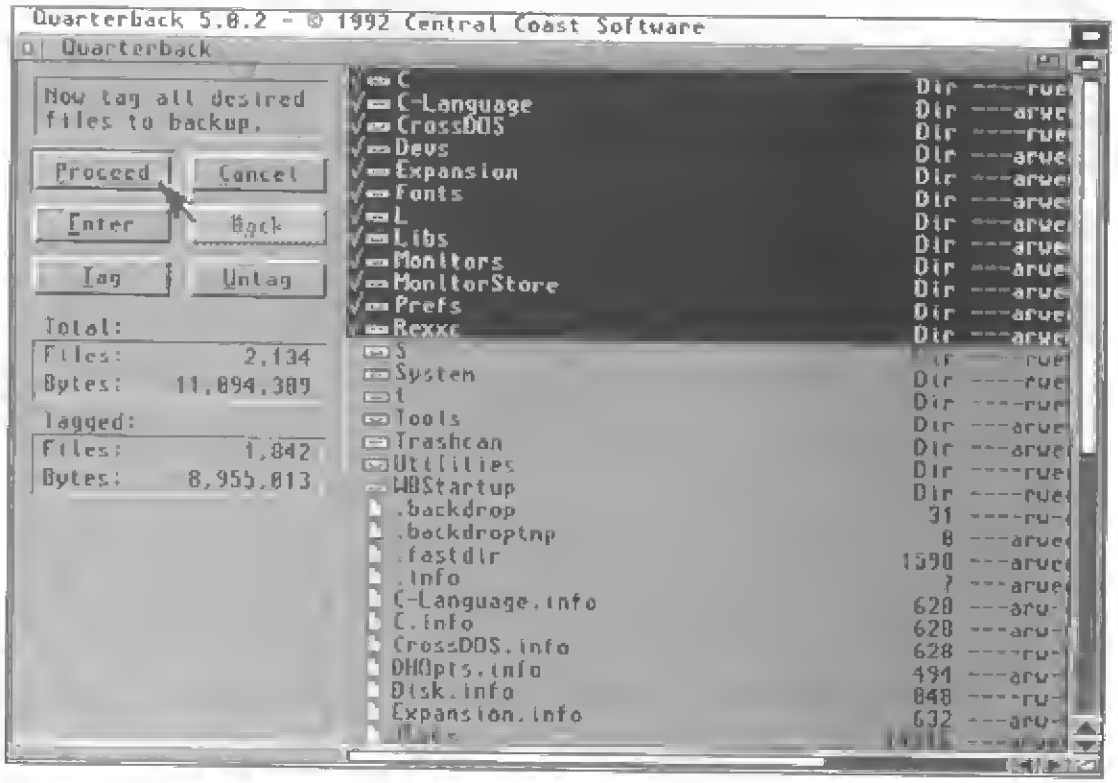
Quarterback 5.0

by Merrill Callaway

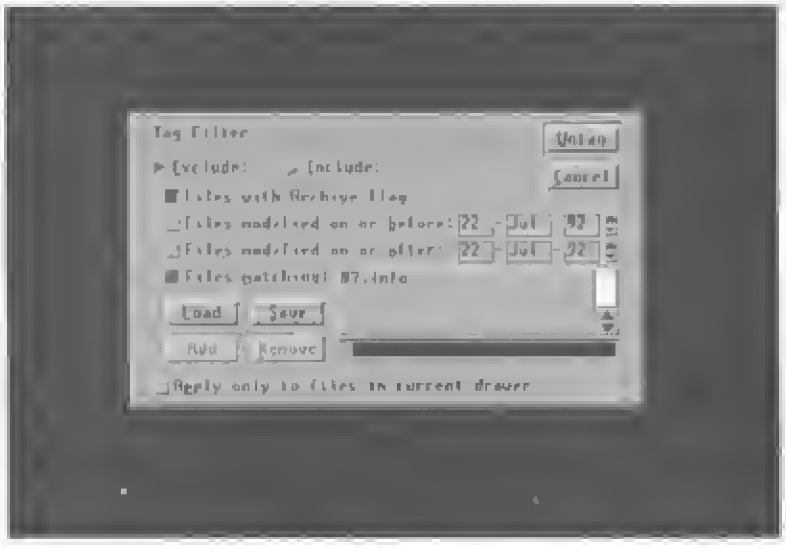
Central Coast Software, a division of New Horizons, Inc., has improved *Quarterback* with release 5.0.2. Quarterback works with all AmigaDOS systems (1.2 to 2.04) and on Amigas 500 to 3000. I tested it on an Amiga 3000 running System 2.04. Quarterback is one of the first and most indispensable programs hard disk users should buy. Like car insurance, a reliable backup utility is an essential, and Quarterback has a deserved reputation for reliability and user friendliness. Quarterback can save you a world of hurt if your hard disk crashes. Once, I was "optimizing" my hard disk when a lightning strike miles away tripped a power company circuit breaker. The lights were only off for 10 seconds, but that was long enough to commit me to a jolt of panic, followed by several days of restoring the hard disk I hadn't backed up recently. You will be thankful for the user friendliness of Quarterback, because restoring a hard disk is a stressful task; ask anyone who has endured it. That accident made a believer out of me... in backing up frequently, and in using an Uninterruptible Power Supply (I use the American Power Conversions "Smart 600").

Uses of a Backup Utility

What does a backup program do? It safeguards your data by copying the entire contents of your hard disk to floppies, tape, removable media (flopticals or Syquest disks), or even another hard drive. Theoretically, you could just copy every file on your hard disk to a corresponding file on a floppy, and then you'd be able to restore (copy back to the hard disk) any files that were corrupted by a crash, mistakenly deleted, or whatever. What do you do, however, with those graphic files that are larger than a floppy's capacity? That's one of the reasons to have Quarterback. Not only can Quarterback copy files larger than one floppy disk to several floppies, but it



Quarterback 5.0's main interface screen.



copies all files so as to maximize efficiency. Quarterback has its own disk format that ignores the disk capacity—it simply fills as much of the disk as it can and then asks for another disk. Quarterback's proprietary format splits and links files across one or more floppy disks, should the end of the disk come before the file has finished copying. Thus, there is no wasted floppy disk space.

The Interface

Since New Horizons took over Central Coast Software, they have made the already good Quarterback interface truly excellent. No one makes a nicer looking interface than New Horizons. The new Quarterback has crisp icons and tasty but businesslike colors. The interface functions well, too. One slick feature is that pressing the "Alt" key changes the buttons to "alternate" functions. For instance, the "Alt" key changes the "Tag" button to a "Tag All" button. This prevents a cluttered interface, yet you must remember only "Alt" to get the alternate set. All functions have keyboard and function key shortcuts, too. A summary is given in the back of the manual.

File Catalogs

Quarterback builds a catalog of files in the disk partition or drawer you select, and makes it easy to manipulate this catalog (running into the thousands of files on a large hard disk). You can "tag" (select) which files and directories are to be backed up or restored, because you will not want to process all files every time. How do you know which files you should back up? It makes good sense to back up only those files that have changed since the last time you backed up your disk. This is called an "incremental backup."

The Archive Bit

AmigaDOS uses an "archive bit" that may be set by the AmigaDOS "protect" command or by clicking on the box next to "archived" inside a file icon information window. You may ask Quarterback to set (or *not* set) the archive bit after a backup, but the archive bit is always turned off *automatically* by AmigaDOS if the file is ever saved or written to. Quarterback recommends you choose and stick to one of two schemes for backup. Both schemes I and II start with a monthly "full backup" of every file on your hard disk to a set of disks called a "full backup set," in which you get Quarterback to *set* the archive bits of *all* files. Step 2 differs in each scheme.

In Scheme I, you make a weekly "incremental backup" to an "incremental set" of disks (different from the full backup set). Use the *same* "incremental set" of disks each week, and *do not* have Quarterback set the archive bit.

In Scheme II, you use a *different* "incremental set" of disks each week and get Quarterback to *set* the archive bit. Each scheme has advantages and disadvantages, as the manual explains in greater detail. Scheme I uses only two sets of disks, but the incremental backups take longer, whereas Scheme II needs multiple sets of disks and takes a shorter time to back up.

Backup Media

If you have a large hard drive, a large number of floppies will be needed to back it up. Quarterback now supports streaming tape (and QB may control features such as rewinding the tape). It also supports removable media such as 20 MB Flopticals, and 44 or 88 MB Syquest removable cartridges. Quarterback allows you to set the I/O buffer size for removable media backup, so if you have the memory to allocate, it can really fly!

Compression

Quarterback can compress your files by almost 50 percent. Compression ratios depend on whether the file is text, executable, or whatever. You may select from 12- to 16-bit compression. The higher number represents more efficient compression at the expense of more memory and time required to execute. On my 25 MHz A3000, backing up 11.8 MB of mostly executable programs at 16-bit compression took 24 minutes and used 9 floppies (13 without compression). Compression time on an unaccelerated Amiga can be quite long.

Report Generation

Quarterback will generate an Archive Report for you and save it to disk or print it if you wish. The report format and what information to include is user settable. You may even add Session Information of your own.

Other Features

When Quarterback generates the catalog of files to back up, or loads a catalog from a backup set for restoring, it automatically "tags" all files in the catalog, as only "tagged" files will be backed up or restored. You may override this feature with "tag filters." We have seen that you may use this feature to exclude those file with the archive bit set. You may prefer to include or exclude files by date (after or before user-supplied dates), or by AmigaDOS pattern matching. The Quarterback Tag Filter accomplishes this. You may also make and save an "automatic tag filter." Specific filters may be given names, and saved and loaded through the Tag Filter menu.

Quarterback has a set of 45 ARexx commands which control Quarterback's functions remotely. You may write macros to do fully automatic backups (with tape or other media with large enough capacity). Shifted function keys may be used to launch 10 macros, and Quarterback's Macro menu may be edited to show your macro names.

Password protection is available, as well as a rich list of preference settings. Under 2.04, Quarterback supports Appicons and the Tools menu on Workbench. The manual is well written, illustrated, and clear. It lacks an index, however, a shameful omission in *any* manual.

Problems

I found two errors . On my A3000 with four floppy drives, two of them outside, Quarterback gets "write errors" on DF3: the last floppy drive "daisy chained" to drive DF2: (which is connected to the Amiga). Physically swapping the two external drives proved that the errors were coming from Quarterback, as it was always address "DF3:" which got errors, regardless of which actual drive unit was attached.

Quarterback puts an entry in the Workbench Tools menu. Quarterback doesn't like *ToolsX* by Steve Tibbets, a program that allows you to put *any* program in the tools menu. Try to run more than one copy of Quarterback. They start OK. They close OK. Try to start Quarterback again; the system crashes. This doesn't happen if ToolsX is not running or if a second copy of Quarterback never starts. I'd like to see the capacity to turn off the Tools menu feature in Quarterback.

Conclusions

Quarterback is a superb backup program—attractive, reliable, and intuitive. As the song goes, "Backing up is so hard to do..." but Quarterback makes this chore as pleasant as it can get.

Quarterback 5.0
Central Coast Software
P.O. Box164287
Austin, TX 78746
(512) 328-6650

Please write to:
Merrill Callaway
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

Developer's Tools

There's Gold In Them There Disks

There is a treasure of applications, utilities, information, and even complete manuals on programming the Amiga in one place—The Fred Fish Collection. The consistent high quality, detailed explanations, and predominately top-notch contents have made the Fred Fish Collection the first resource of thousands of Amiga users throughout the world.

Mr. Fish has won numerous awards as well as the thanks and acclaim of the entire Amiga community in his efforts to be the master software anthologist for the Amiga. With disk number 720 just released, Fred's efforts are nothing less than astounding. This huge collection of software contributed by Amiga users and developers from all over the globe now numbers more than 3,000 individual programs. The reason for the anthology was immediately apparent to Fred back in 1985 when he began to collect it. The Amiga needed software, and the only way that the software could be developed was to share information. He created a few disks for himself and then shared them at a local Amiga user group meeting. His fellow Amiga users were extremely grateful and have been so ever since. (To hear more about Fred and the early days of the Amiga, please read Steven King's interview with Fred Fish in the October 1992 issue of *Amazing Computing For The Commodore Amiga.*)

Aside from the wide variety of programs, demonstration software, animations, and more available in the Fred Fish Collection, there is also a large selection of programming tools and utilities. Dr. Patrick Bailey, the author of the unique catalog system for Fred Fish disks called Cat Fish, has defined 15 different categories of programming tools and examples under the main title of programming, as well as 32 separate categories in the utility section. This is in a field of 17 major categories with over 140 total individual subcategories. While not every category or subsection may be of interest to each programmer's work, the combined library is an arsenal of important tools.

For this article, we have included some of the important files added to the library in the past few months. We have chosen from the programming and utilities sections as defined by the Cat Fish system. While we do not have room to reprint the entire collection of programs from the 720 disks now in the library, this quick list will give *AC's TECH* readers some idea of the potential available in this collection. The descriptions are edited versions of the description files written by the master Amiga software anthologist himself, Fred Fish, and are available on the assorted disks.

PROGRAMMING: ASSEMBLER

ADev11 A complete development system for Motorola's 68HC11 processor, including a macro assembler, linker, librarian, downloader, and disassembler. Supports multiple source file and multiple relocatable segments per file. Binary only. Author: Stan Burton 717

EZAsm Combines 68000 assembly language with parts of C. Produces highly optimized code. Uses C-like function calls (supports all 2.04 functions), braces, 'else', '.fd' support, and much more. Comes bundled with A68k and Blink, for a complete programming environment. Version 1.7. Includes example source and executable files. Binary only. Author: Joe Siebenmann 699

IFFLib An easy-to-use Amiga library which gives you some powerful routines for dealing with IFF files, especially ILBM files (pictures), ANIM files (animations), and 8SVX files (digitized sounds). It is written completely in assembler and is just 3K. Includes source and binaries for several example programs that use the library. Version 22.2. Binary only. Author: Christian A. Weber 674

Ninfo A disassembler for memory, boot blocks, objects, libraries, and executables. Version 2.0, includes source. Author: Tony Preston 662

PROGRAMMING: C

CManual Five-disk distribution of one of the largest collections of documents, examples, and utilities in C for the Amiga. It consists of six manuals, with more than 40 chapters, 175 fully executable examples complete with source code, and several utilities and other goodies. The manuals describe how to open and work with Screens, Windows, Graphics, Gadgets, Requesters, Alerts, Menus, IDCMP, Sprites, VSprites, AmigaDOS, Low Level Graphics Routines, etc. They also explain how to use your C Compiler and give you important information about how the Amiga works and how your programs should be designed. When unpacked, the manuals and examples nearly fill up twelve standard Amiga floppies. Version 3.0. Because of its size, it is distributed on five library disks, 691 through 695. Author: Anders Bjerin.

DebugUtils Some tools for use in debugging applications. When used with Enforcer and Mungwall, they comprise an integrated set that gives the programmer a powerful and flexible debugging aide. Author: Mark Porter 663

Flex A replacement for the UNIX lex (lexical analyzer generator) program that is faster than lex, and freely redistributable. Lexical analyzer generators are generally used in combination with parser generators (such as yacc or bison), to generate front ends for language compilers and other tools. Version 2.3.7. Includes source. Author: Jef Poskanzer, Vern Paxson, et al 704

FontConverter Converts standard font files into C code structures that can be included directly in your program. Probably most useful for people writing programs that take over the machine and thus do not have access to the standard fonts directory. Includes source. Author: Andreas Baum 644

Hextract A complete header file reference. Definitions, structures, structure members and offsets, flag values, library contents, function definitions, registers, library offsets, etc. The data from a set of V1.3 Amiga and Lattice header files is packed into the included file 'headers.z' for immediate reference by Hextract. Version V1.1, freeware. Includes part source. Author: Chas A. Wyndham 674

Indent A C source code formatter/indenter. Especially useful for cleaning up inconsistently indented code. Version 1.3. Includes source. Author: Various, Amiga port by Carsten Steger 672

Indent A C source code formatter/indenter. Especially useful for cleaning up inconsistently indented code. Version 1.4. Includes source. Author: Various, Amiga port by Carsten Steger 702

PROGRAMMING: Miscellaneous

F2C A program that translates Fortran 77 source into C or C++ source. F2C lets one portably mix C and Fortran, and makes a large body of well-tested Fortran source code available to C environments. Amiga port done for SAS/C 5.10B, and includes libraries for use with SAS/C. Includes full source in C. Author: S. I. Feldman, David M. Gay, Mark W. Maimone, and N. L. Schryer; Amiga port by Martin Hohl 675

MPE A compiler tool for users of the M2amiga programming environment. MPE does the same job better than your batch file. You can do everything with the mouse or the right Amiga key. With this Modula-2 Programming Environment you can compile, link, and run your program. When there is an error, the editor is started automatically. You can set all switches for M2C, M2L, and M2Make. Version 1.17. Binary only. Author: Marcel Timmermans 703

SANA The official Commodore developer information package for the SANA-II Network Device Drivers. Includes the SANA-II spec, readme files, SANA-II drivers for Commodore's A2065 (Ethernet) and A2060 (ARCNET) boards, docs and includes. Author: Commodore-Amiga Networking Group 673

TalinCode A bunch of source code for demos, tests, and experiments, that the author wrote over a period of eight years, mostly for recreation or for general R&D for projects that never materialized. Includes 3-D techniques, a maze generator, logarithms, basic utility functions, DOS functions, random numbers, and much more. Includes source, mostly in assembly code. Author: David Joiner 716

UTILITIES: ARexx

RexxHostLib This is a shared library package to simplify the ARexx host creation/management procedure. Rexx-message parsing is also included, making it possible to control ARexx from programs such as AmigaBASIC (can you imagine AmigaBASIC controlling AmigaTeX?). Version 37.1. Updated for use with Kickstart 2.0. Includes source in 'C' and assembly language. Author: Olaf 'Olsen' Barthel 682

UTILITIES: Benchmarks

DiskSpeed A disk speed testing program specifically designed to give the most accurate results of the true disk performance of the disk under test. Automatically updates and maintains an ASCII database of disk results for tested disks. Version 4.2. Includes source in C. Author: Michael Sinz 665

UTILITIES: Calculators

ICalc A powerful calculator with many features, including user-defined variables and functions, C-style programming constructs, complex number calculations and more. Has comprehensive instructions, and numerous examples. Version 2.1a. Binary only, source available from author. Author: Martin W. Scott 713

UTILITIES: CLI and CLI Shells

AUSH A new command line interpreter, designed to replace the CBM shell. Features include file name completion, pattern expansion, expression computation, command history, for...done loops, and much more. Almost fully compatible with ARP or Commodore shells. Version 1.42, AmigaDOS 2.04 support. Requires arp.library under 1.3. Binary only. Author: Denis Gounelle 706

GMC A console handler with command line editing and function key support. GMC provides extended command line editing, function key assignment in four levels, extended command line history, online help for functions in the handler, and an iconify function. Also includes an output buffer (dump to printer and window), filename completer, script function, undo function, prompt beeper, pathname in window title, close gadget for KS 2.0, etc. Version 9.13. Shareware, binary only. Author: Goetz Mueller 683

SKsh A ksh-like shell for the Amiga. Some of its features include command substitution, shell functions, aliases, local variables, emacs and vi style command line editing, I/O redirection, pipes, UNIX style wildcards, a large variety of commands, and coexistence with scripts from other shells. Well documented. Version 2.0. New features include real pipes, AmigaDOS 2.04 support, enhanced ARexx handling, and more. Binary only. Requires AmigaDOS 2.04. Author: Steve Koren 672

UTILITIES: CLI Commands

CLIExe A Xicon style program. It allows you to execute a script from WB and is completley CLI compatible, because it is a CLI. Can use a real script file or take commands in its own TOOLTYPES. Version 1.1. Includes source in C. Author: Sylvain Rougier 649

Install A replacement for the AmigaDOS Install command, with an Intuition front end Version 1.1. Includes source in assembly. Author: David Kinder 643

Scan Program to scan file contents for matches to one or more specified patterns. Claimed to search hard drives twice as fast as the best search programs currently available, and RAM drives five times faster than other programs. Can optionally scan the contents of files in LZH and LHA archives. Supports searching for multiple patterns simultaneously. Other features include extensive wildcard support, optional inverted pattern matching, recursive directory scanning, line search highlights of matching words with selectable color, and more. Version 1.0. Includes source. Author: Walter Rothe 670

UTILITIES: Dir/File Displayers

BrowserII A 'Programmer's Workbench'. Allows you to easily and conveniently move, copy, rename, and delete files and directories using the mouse. Also provides a method to execute either Workbench or CLI programs by double-clicking them or by selecting them from a ParM like Menu with lots of arguments. Version 2.04. Binary only. Author: Sylvain Rougier, Pierre Carrette 649

DirWork A fast, small, efficient, DirUtility. Configurable options and buttons, as well as all the usual features. Comes with external configuration editor. Version 1.51. Shareware, binary only. Author: Chris Hames 670

Du A very small (only 932 bytes) program to display the total disk space used by a directory and all its sub-directories. Version 2.5. Enhancements include wildcards, totals, clearer output, plus the program can be made resident. Requires Kickstart 2.0. Includes source in assembler. Author: Stuart Mitchell 701

SID A very comprehensive directory utility for the Amiga that supports at least a couple of dozen different commands for operating on files. Version 2.0. Binary only. Author: Timm Martin 651

UTILITIES: Disk Utilities

AmiBack Demo version of Ami-Back v2.0, a nice backup utility for the Amiga. Features include backup to any AmigaDOS compatible device (such as floppies, removable hard disks, fixed media hard disk, and tape drives), compression, no copy protection, configuration files, complete backups, incremental backups, selective backups, file exclusion filter, setting of archive bit, password protected backups, online help, AREXX support, etc. Demo version does not have restore. Version 2.0a. Binary only. Author: MoonLighter Software 682

EraseDisk A small, fast program used to erase a disk by setting all bits on the disk to zero. Version 0.92. Binary only. Author: Otto Bernhart 650

FreeCopy FreeCopy is unlike most copiers in that it does not actually copy disks. It removes the protection so disks can easily be backed up with almost any program, and in some cases be installed on your hard drive. Version 1.8. Public domain, binary only. Author: Greg Pringle 685

Zoom A fast and efficient floppy disk archiving utility based on the data compression/decompression algorithms used by lh.library. Has an Intuition and a Shell interface, fully supports Kickstart 2.0, is able

to add texts and notes to archived output files, knows 274 different bootblock viruses, includes a number of compression parameters (such as encryption of the output file) and a lot more. Version 5.4. Binary only. Author: Olaf 'Olsen' Barthel 682

UTILITIES: File Compressions

LhA A very fast archiver that is compatible with MS-DOS LhArc V1.13 and LHA V2.13, as well as the Amiga LhArc. LhA is very memory efficient, has been written with stability and reliability in mind, has carefully optimized compression and decompression routines, is multitasking reentrant and pure, handles multiple volume archives (registered version only), and more. Version 1.32. Shareware, binary only. Author: Stefan Boberg 715

P-Compress A compression program that produces smaller files faster than any other current general-purpose cruncher, using LZH compression algorithms. Can handle single files, whole drawers, disks, or selected files or types of files within drawers and disks. Includes compression and decompression object files which can be linked to your own programs to allow them to access and output data in LZH format. Version 2.3. Freeware, binary only. Author: Chas A.Wyndham, LZH code by Barthel/Krekel 650

P-FixLib A new P-Suite utility that diverts calls to DOS library so that P-Compressed files are decompressed before being opened or executed. Any type of file, including icons, executables, libraries, fonts, texts, etc. may be compressed. Effectively doubles the capacity of your disks. Version 1.2, freeware, binary only. Author: Chas A.Wyndham 650

P-Writer A text editor with special facilities for inserting text color and style changes and for preparing illustrated texts for P-Reader. Version 3.3. Freeware, binary only. Author: Chas A. Wyndham 674

PPLib A shared library to make life easy for people who wish to write programs that support PowerPacker. Loading crunched files from C or assembly is made fast, short, and easy. Release 1.5. Includes example source. Author: Nico Francois 678

UTILITIES: Icons and Pointers

Icons Some Workbench 1.3 icons with a Workbench 2.0 3D look. They also look pretty good under 2.0 when simply run through one of the many icon remapping tools available. Author: L. Guzman 708

MakeIcon Allows you to create any of the Workbench 2.0 default icons for anything, disks, projects, drawers, the works. Designed for people who work from the shell making disks that will ultimately run from the Workbench. Requires Workbench 2.0. Binary only. Author: Robert Lang 719

UpdateIcon A tool to add icons to files and drawers which do not yet have icons attached, to update the default tools and to reset the positions of icons. Very handy for installation scripts. Requires Kickstart and Workbench 2.04 (or higher). Version 1.0, includes source in C. Author: Olaf 'Olsen' Barthel 688

UTILITIES: Postscript Uses

Post An excellent PostScript interpreter for the Amiga which implements the full Adobe language. Supports type 1 and type 3 fonts, screen output, file output, and printer output. Requires Arp library V39+ and ConMan V1.3+ (only under AmigaDOS 1.3). Version 1.7. Includes source in C. Author: Adrian Aylward 669

UTILITIES: Printing

APrf A freely redistributable printing utility for the Amiga. Features include a full Intuition interface, preview function, page selection, margin setup, line numbering, an ARexx port, a multi-columns mode, 2.04 system release support, and more. Includes both French and English versions. Version 1.30. Binary only. Author: Denis Gounelle 706

DiskPrint A label database which prints and stores disk labels for 3.5 and 5.25 disks. Primarily created as a combined database and print utility for FD disks, it includes easy-to-use label library functions (like printing labels for a whole FD series in one turn or multiple print of one label) and labels for most FD disks which are available within a few mouse clicks. Features include a fast search routine, user-definable label layout, different label sizes, intuition-based disk directory read-in, and a lot more. Very configurable. Works fine with every printer connected to the parallel port and AmigaDOS 1.2, 1.3, and 2.x. Version 3.51. Both English (PAL and NTSC) and German versions. Shareware, binary only. Author: Jan Geissler 685

UTILITIES: Screens and Windows

BootPic BootPic allows you to install nearly any IFF picture that you like in place of the Workbench hand that appears after a reset, and additionally plays a MED-Module. Version 2.1b. Includes source in assembly. Author: Andreas Ackermann 718

WindowTiler A Workbench 2.0 commodity for arranging windows. Comes with many tool types to help customize it. Supports virtual screen users, tiling, cascading, exploding windows, etc. Version 2.1b. Binary only. Author: Doug Dyer 696

UTILITIES: Tasks & Memory Pgms.

ARTM Amiga Real Time Monitor displays and controls system activity such as tasks, windows, libraries, devices, resources, ports, residents, interrupts, vectors, memory, mounts, assigns, locks, fonts, hardware and res_cmds. Version 1.6. Shareware, binary only. Author: Dietmar Jansen and F. J. Mertens 652

Free Display how much free space (bytes or blocks) you have on any or all of your mounted disk volumes. Runs from CLI only. Version 1.06. Free now searches your device list if desired (under AmigaDOS 2.0+ only). Includes source. Author: Daniel J. Barrett 713

JM Job Manager is a utility which extends the AmigaDOS multitasking environment by providing features such as: allocation of CPU cycles in any ratio to multiple CPU bound processes, default task priorities based on task name, task logging, system uptime reports, task CPU use and CPU % reports, task invocation times, and more. JM has very little impact on the system itself. Requires AmigaDOS 2.04 or later. Includes 68000/20 and 68030/40 versions. Version 1.1. New features include better task name detection, and an ARexx port. Binary only. Author: Steve Koren 647

Notify A suite of Rexx programs that can be used to issue messages or run commands automatically on certain days and/or at certain times of day. Facilities are provided for the adding, editing and deleting of messages, and for displaying the times and texts of pending messages. A chime program is included to enable the time to be announced at regular intervals. Version 1.02. Author: Michael Tanzer 652

SysInfo A program which reports interesting information about the configuration of your machine, including some speed comparisons with other configurations, versions of the OS software, etc. This program has been very popular with many users and has been fully updated to include many new functions. Version 2.69. Binary only. Author: Nic Wilson 642

SystemInfo A system configuration display program with an Intuition interface. Recognizes about 80 different product codes and about 40 manufacturer ID's. Displays information about all AutoConfig cards, all mounted drives, vectors, processor types, and other useful information. Version 2.0a, shareware, binary only. Author: Paul Kolenbrander 644

UTILITIES: Text File Displayers

Less A port of a UNIX text file reader. It can use pipes, accepts multiple filenames, and has many convenient positioning commands for forward and backward movement, marking positions, etc. Version 177.4. Includes source. Author: Mark Nudelman, port by Frank Busalacchi 718

View A text displayer with many controls and features including searches, file requestors, jump to editor, etc. Version 2.0. Includes source. Author: Jan Van Den Baard 658

UTILITIES: UNIX

UnixUtils A collection of UNIX-like programs for the Amiga. Includes head, tail, sort, strings, diff and find. The first four are original programs; find is derived from tree by Tomas Rokicki, diff is a port of the GNU version. Includes source. Author: Maurizio Loreti 663

UTILITIES: User Interfaces

Arq Replaces the standard system requesters with nice animated requesters to which you can also attach different sounds. Works under AmigaDOS 1.3 or 2.0 to give all the normal system requesters a nice new look. Version 1.66. Binary only. Author: Martin Laubach, Peter Wleek, and Rene Hexel 665

BlackHole A file deletion utility for v2.04 and greater of the operating system. When run, it puts an appicon on the Workbench screen. Any file/drawer icons that are dropped on it will be deleted. Double clicking on the appicon brings up an options window. Version 1.1, includes source. Author: Alan Singfield 662

GadToolsBox A program that lets you draw/edit GadTools gadgets and menus and then generates the corresponding C or assembly code for you. Version 1.3. Includes source. Author: Jan van den Baard 659

NoDelete This program pops up a requester to alert you of a file deletion being attempted via DeleteFile() and allows you to accept or cancel it. This also pertains to any files you attempt to delete via 'delete'. Version 2.01. Includes source. Author: Uwe Schuerkamp 657

ParM Parameterable Menu. ParM allows you to build menus to run whatever program you have on a disk. ParM can run programs either in WorkBench or CLI mode. This is an alternative to MyMenu which can run only when WorkBench is loaded. ParM can have it's own little window, can attach menus to the CLI window you are running it from, or to the WB menus, just like MyMenu. Version 3.6. Includes source in C. Author: Sylvain Rougier, Pierre Carrette 649

PopUpMenu A small program that makes it possible for you to use pop-up menus with any program that uses standard intuition menus. Version 4.3. Includes source. Author: Martin Adrian 667

WBLink WBLink adds an 'AppIcon' to the Workbench 2.0 screen that creates a link to whatever file or directory is dragged on top of it. This version fixes some bugs and lets the user specify where the icon will be placed. Version 1.10. Includes source. Author: Dave Schreiber 654

UTILITIES: Miscellaneous

Enforcer Detects/protects against illegal memory hits. Compatible with all DOS versions and Amigas (requires a Memory Management Unit or 68030 processor). The low 1K of memory and all areas that are not RAM are protected from CPU reads or writes. ROM is marked as read-only. Version 2.8b, an update to version 2.6f on disk 474. Binary only. Author: Bryce Nesbitt 658

PatchOS Enhances OS 2.04 with three new features: keyboard-shortcuts for menus while a string-gadget is active, use of the star ("*") in AmigaDOS pattern matching, and input of any char by typing its ASCII-code on the numeric pad. Requires at least AmigaDOS 2.04. Implemented as a commodity. Version 1.00. Includes documentation in German and English language. Author: Hartmut Stein / Bernstein Zirkel Softworks 706

UTILITIES: Virus Check & Fixing

AntiCicloVir A link virus detector that detects 25 different such viruses. Version 1.5. Shareware, binary only. Author: Matthias Gutt 710

BootX An easy-to-use boot, file and link virus killer. For use with Kickstart 2.0 only. Has lots of options to detect and kill Amiga viruses, extensive manual, locale support and AmigaGuide online help. Version 5.00. Binary only. Author: Peter Stuer 703

LVD A first defense utility against file and linkviruses. It patches the LoadSeg vector(s) and checks every executable that comes along. Recognizes 33 file or so linkviruses. Version 1.73. Binary only. Author: Peter Stuer 703

VirusChecker A virus checker that can check memory, disk bootblocks, and all disk files for signs of most known viruses. Can remember nonstandard bootblocks that you indicate are OK and not bother you about them again. Includes an ARexx port. Version 6.06. Binary only. Author: John Veldthuis 680

More Information:

For a complete printed list of the Fred Fish Collection's programs, please see the latest issue of *AC's GUIDE To The Commodore Amiga*. In addition, each month *Amazing Computing For The Commodore Amiga* lists the latest disks received at press time.

Dr. Peter Bailey's Cat Fish Catalog is available in the latest format for $10 from ASCITEC, Amiga Science and Tech Users' Group, P. O. Box 201, Los Altos, CA 94023-0201, (415) 960 9399.

Should You?

Amaze Them Every Month!

Amazing Computing For The Commodore Amiga is dedicated to Amiga users who want to do more with their Amigas. From Amiga beginners to advanced Amiga hardware hackers, AC consistently offers articles, reviews, hints, and insights into the expanding capabilities of the Amiga. *Amazing Computing* is always in touch with the latest new products and new achievements for the Commodore Amiga. Whether it is an interest in Video production, programming, business, productivity, or just great games, AC presents the finest the Amiga has to offer. For exciting Amiga information in a clear and informative style, there is no better value than *Amazing Computing* .

A Guide For Every Amiga User.

Give the Amiga user on your gift list even more information with a SuperSub containing *Amazing Computing* and the world famous *AC's GUIDE To The Commodore Amiga*. *AC's GUIDE* (published twice each year) is a complete listing of every piece of hardware and software available for the Amiga. This vast reference to the Commodore Amiga is divided and cross referenced to provide accurate and immediate information on every product for the Amiga. Aside from the thousands of hardware and software products available, *AC's GUIDE* also contains a thorough list and index to the complete Fred Fish Collection as well as hundreds of other freely redistributable software programs. No Amiga library should be without the latest AC's GUIDE.

More TECH!

AC's TECH For The Commodore Amiga is an Amiga users ultimate technical magazine. *AC's TECH* carries programming and hardware techniques too large or involved to fit in *Amazing Computing*. Each quarterly issue comes complete with a companion disk and is a must for Amiga users who are seriously involved in understanding how the Amiga works. With hardware projects such as creating your own grey scale digitizer and software tutorials such as producing a ray tracing program, *AC's TECH* is the publication for readers to harness their Amiga to fulfill their dreams.

Why Not?

Amazing Computing and AC's TECH—

So many reasons to subscribe, so many ways to save...

1-800-345-3360

High Resolution Output

from your AMIGA™ DTP & Graphic Documents

You've created the perfect piece, now you're looking for a good service bureau for output. You want quality, but it must be economical. Finally, and most important...you have to find a service bureau that recognizes your AMIGA file formats. Your search is over. Give us a call!

We'll imageset your AMIGA graphic files to RC Laser Paper or Film at 2400 dpi (up to 154 lpi) at a extremely competitive cost. Also available at competitive cost are quality Dupont ChromaCheck™ color proofs of your color separations/films. We provide a variety of pre-press services for the desktop publisher.

Who are we? We are a division of PiM Publications, the publisher of *Amazing Computing for the Commodore AMIGA*. We have a staff that *really* knows the AMIGA as well as the rigid mechanical requirements of printers/publishers. We're a perfect choice for AMIGA DTP imagesetting/pre-press services.

We support nearly every AMIGA graphic & DTP format as well as most Macintosh™ graphic/DTP formats.

For specific format information, please call.

For more information call 1-800-345-3360

Just ask for the service bureau representative.

INTRODUCING
THE WAVE
OF THE FUTURE

The SAS/C® Development System, Version 6

Ride the wave of the future with our new release of the SAS/C Development System—Version 6. It's fast, flexible, and powerful, offering you new ways of producing the most efficient code for the Amiga®. Explore a whole new world of development capabilities with these new Version 6 features and enhancements:

- an integrated environment
- fully ANSI-compliant compiler and libraries
- improved CodeProbe debugger
- new global and peephole optimizers
- greatly enhanced error and warning messages
- all new documentation
- increased AREXX support
- online help
- free technical support.

Use the development system that Commodore® relied on to create Release 2 of the Commodore Amiga operating system and uncover the treasures it holds! To order Version 6 of the SAS/C Development System or for a free brochure, call SAS Institute at 919-677-8000, extension 7001.

If you're currently using another commercial C compiler, call us for information on our special trade-in offer!

SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513